Cover page

Installable File Systems

For OS/2 Version 2.0

OS/2 File Systems Department

PSPC Boca Raton, Florida

Translated to IPF by Andre Asselin

Version 0.1

July 1993

Disclaimer

Changes

This document is based on the February 17, 1992 version of the printed IFS documentation.

July, 1993

Installable File System Mechanism

The OS/2 Installable File System (IFS) Mechanism supports the following:

oCoexistence of multiple, active file systems in a single PC
oMultiple logical volumes (partitions)
oMultiple and different storage devices
oRedirection or connection to remote file systems
oFile system flexibility in managing its data and I/O for optimal performance
oTransparency at both the user and application level
oStandard set of File I/O API
oExisting logical file and directory structure
oExisting naming conventions
oFile system doing its own buffer management
oFile system doing file I/O without intermediate buffering
oExtensions to the Standard File I/O API (FSCTL)
oExtensions to the existing naming conventions
oIOCTL type of communication between a file system and a device driver

----

Installable File System Overview

System Relationships

Installable File System (IFS) Mechanism defines the relationships among the operating system, the file systems, and the device drivers. The basic model of the system is represented in Figure 1-1.

```
          File System Request Router
```

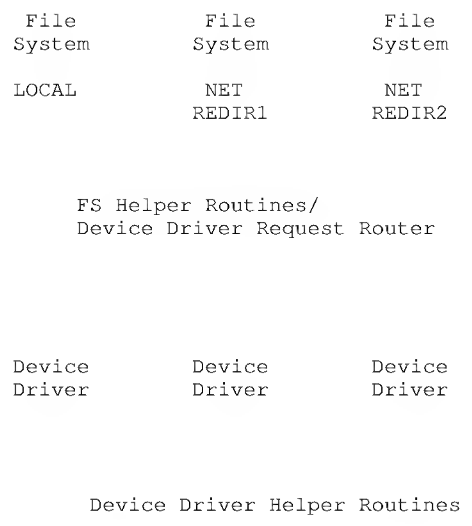

Figure 1-1. System relationships for Installable File Systems

The file system request router directs file system function calls to the appropriate file system for processing.

The file systems manage file I/O and control the format of information on the storage media. An installable file system (FS) will be referred to as a file system driver (FSD).

The FS Helper Routines provide a variety of services to the file systems.

The device drivers manage physical I/O with devices. Device drivers do not understand the format of information on the media.

File I/O API

Standard file I/O is performed through the Standard File I/O API. The application makes a function call and the file system request router passes the request to the correct file system for processing. See Figure 1-2.

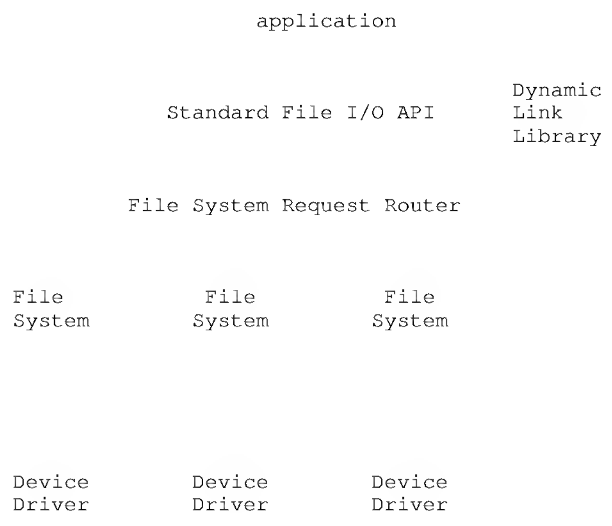

Figure 1-2. Standard File I/O

New API may be provided by a file system to implement functions specific to the file system or not supplied through the standard file I/O interface. New API are provided in a dynamic link library that uses the DosFSCtl standard function call to communicate with the specific file system (FSD). See Figure 1-3.

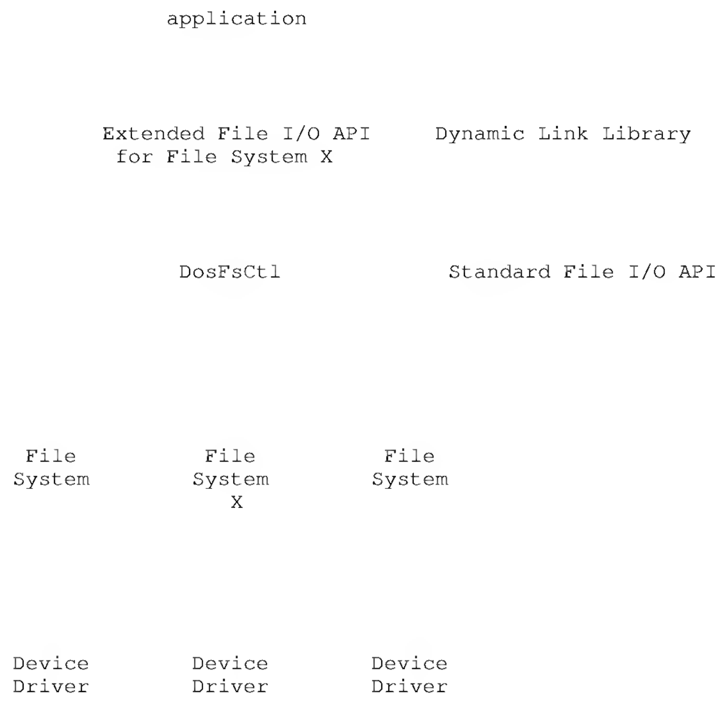

Figure 1-3. Extended File I/O

Buffer Management

In 2.0 the FAT buffer management helpers were removed because of lack of use by any 1.x FSD. FSDs should handle all buffer/cache management themselves.

The FSD moves all data requiring partial sector I/O between the application 's buffers and its cache buffers. The FS helper routines initiate the I/O for local file systems.

Volume Management

Volume management (that is, detecting when the wrong volume is mounted and notifying the operator to take corrective action) is handled directly through OS/2 and the device driver. Each FSD is responsible for generating a volume label and 32 -bit volume serial number. These are stored in a reserved location in logical sector zero at format time. Because an FSD is the only system component to touch this information, an FSD is not required to store it in a particular format. OS/ 2 calls the FSD to perform operations that might involve it. The FSD is required to update the volume parameter block (VPB) whenever the volume label or serial number is changed.

When the FSD passes an I/O request to an FS helper routine, the FSD passes the 32-bit volume serial number and the user's volume label (through the VPB) . When the I/O is performed, OS/2 compares the requested volume serial number with the current volume serial number it maintains for the device. This is an in -storage test (no I/O required) performed by checking the drive parameter block 's (DPB) VPB of the volume mounted on the drive. If unequal, OS/2 signals the critical error handler to prompt the user to insert the volume having the serial number and label specified.

When OS/2 detects a media change in a drive, or the first time a volume is accessed, OS/2 determines which FSD is responsible for managing I/O to that volume. OS/2 allocates a VPB and polls the installed FSDs (by calling the FS_MOUNT entry point) until an FSD indicates that it does recognize the media.

Note: The FAT FSD is the last in the list of installed FSDs and acts as the default FSD when no other FSD recognition takes place.

Connectivity

There are two classes of file system drivers:

oFSDs that use a block device driver to do I/O to a local or remote device . These are called local file systems.
oFSDs that access a remote system without a block device driver. These are called remote file systems.

The connection between a drive letter and a remote file system is achieved through a command interface provided with the FSD (FS_ATTACH).

When a local volume is first accessed, OS/2 sequentially asks each installed FSD to accept the media, by calling each FSD's FS_MOUNT entry point. If no FSD accepts the media, it is then assigned to the default FAT file system. Any further attempt that is made to access an unrecognized media, other than by FORMAT, results in an 'Invalid media format' message.

When a volume has been recognized, the relationship between drive, FSD, volume serial number, and volume label is remembered. The volume serial number and label are stored in the volume parameter block (VPB). The VPB is maintained by OS /2 for open files (I/O based on file-handles), searches, and buffer references . The VPB represents the media.

Subsequent requests for a volume that has been removed require polling the installed FSDs for volume recognition by calling FS_MOUNT. The volume serial number and volume label of the VPB returned by the recognizing FSD and the existing VPB are compared. If the test fails, OS/2 signals the critical error handler to prompt the user for the correct volume.

The connection between media and VPB is remembered until all open files on the volume are closed and search and cache buffer references are removed. Only volume changes cause a redetermination of the media at the time of next access.

IPL Mechanism

A primary DOS disk partition (type 1, 4, or 6) may be used to boot the system . The code for FSDs may reside in any partition readable by a previously installed FSD. An IFS partition must be a type 7 partition.

The OS/2 boot volume includes the following: Bootrecord and basic file system. The root directory of this volume will contain a mini-file system in OS2BOOT, a kernel loader in OS2LDR, the OS/2 kernel in OS2KRNL, and the CONFIG.SYS file.

Device drivers and FSDs are loaded in the order they appear in CONFIG.SYS and are considered elements of the same ordered set. Therefore, both device drivers and FSDs may be loaded from installed file systems as long as they are started in the proper order. For example:

```
DEVICE = c:\diskdriv.sys
REM Block device D: is now defined. (diskdriv.sys controls this.)
IFS = c:\fsd\newfsl.fsd
REM If we assume that D: contains a fixed newfsl type partition,
REM then we're now ready to use D: to load the device driver and
REM FSD for E:.
DEVICE = d:\root\dev\special.dev
REM Block device e: is now defined.
IFS = d:\root\fsd\special.fsd
REM E: can now be read.
DEVICE = e:\music
```

OS/2 Partition Access

Access to the OS/2 partition on a bootable, logically partitioned media is through the full OS/2 function set. See *OS/2 Version 2.0 Physical Device Driver Reference* for a detailed description of the disk partitioning design.

Permissions

There are no secure file system clients identified for OS/2 Version 2.0 incorporating the IFS architecture.

File Naming Conventions

See *OS/2 Version 2.0 Programming Guide* for a detailed description of OS/2 Version 2.0 file naming conventions.

Meta Character Processing

See *OS/2 Version 2.0 Programming Guide* for a detailed description of OS/2 Version 2.0 meta character processing.

FSD Pseudo-character Device Support

A pseudo-character device (single file device) may be redirected to an FSD . The behavior of this file is very similar to the behavior of a normal OS/2 character device. It may be read from (DosRead) and written to (DosWrite). The difference is that the DosChgFilePtr and DosFileLocks functions can also be applied to the file. The user would perceive this file as a device name for a non-existing device. This file is seen as a character device because the current drive and directory have no effect on the name. That is what happens in OS/2 today for character devices.

The format of an OS/2 pseudo-character device name is that of an ASCIIZ string in the format of an OS/2 file name in a subdirectory called \DEV\. The pseudo device name XXX is accessible at the API level (DosQFSAttach) through the path name '\DEV\XXX'.

Family API Issues

Since the IFS Mechanism is not present in any release of DOS, FAPI will not be extended to support the new interfaces.

FSD Utilities

FSD Utility Support

Each FSD is required to provide a single .DLL executable module that supports the OS/2 FORMAT, CHKDSK, SYS, and RECOVER utilities. The FS-supported executable will be invoked by these utilities when performing a FORMAT, CHKDSK, SYS, or RECOVER function for that file system. The command line that was passed to the utility will be passed unchanged to the FS-specific executable.

The procedures that support these utilities reside in a file called U<fsdname >.DLL, where <fsdname> is the name returned by DosQFSAttach. If the file system utility support .DLL file is to reside on a FAT partition, then <fsdname> should be up to 7 bytes long.
FSD Utility Guidelines

The FSD utility procedures are expected to follow these guidelines:

oNo preparation is done by the base utilities before they invoke the FSD utility procedure. Therefore, base utilities do not lock drives, parse names, open drives, etc. This allows maximum flexibility for the FSD.

oThe FSD utility procedures are expected to follow the standard conventions for the operations that they are performing, for example, /F for CHKDSK implies 'fix'.

oThe FSD procedures may use stdin, stdout, and stderr, but should be aware that they may have been redirected to a file or device.

oIt is the responsibility of the FSD procedures to worry about volumes being changed while the operation is in progress. The normal action would be to stop the operation when such a situation is detected.

oWhen the FSD procedures are called, they will be passed argc, argv, and envp , that they can use to determine the operations.

oFSD procedures are responsible for displaying relevant prompts and messages .

oFSD utility procedures must follow the standard convention of entering the target drive as specified for each utility.

FSD Utility Interfaces

All FSD utility procedures are called with the same arguments:

```
int far pascal CHKDSK(int argc, char far * far *argv,
char far * far *envp);

int far pascal FORMAT(int argc, char far * far *argv,
char far * far *envp);

int far pascal RECOVER(int argc, char far * far *argv,
char far * far *envp);

int far pascal SYS(int argc, char far * far *argv,
char far * far *envp);
```

where argc, argv, and envp have the same semantics as the corresponding variables in C.

Extended Attributes

Extended attributes (EAs) are a mechanism whereby an application can attach information to a file system object (directories or files) describing the object to another application, to the operating system, or to the FSD managing that object.

EAs associated with a file object are not part of a file object's data, but are maintained separately and managed by the file system that manages that object.

Each extended attribute consists of a name and a value. An EA name consists of ASCII text, chosen by the application developer, that is used to identify a particular EA. EA names are restricted to the same character set as a filename. An EA value consists of arbitrary data, that is, data of any form. Because of this OS/2 does not check data that is associated with an EA.

So that EA data is understandable to other applications, conventions have been established for:

oNaming EAs
oIndicating the type of data contained in EAs

In addition, a set of standard EAs (SEAs) have been defined. SEAs define a common set of information that can be associated with most files (for example, file type and file purpose). Through SEAs, many applications can access the same , useful information associated with files.

Applications are not limited to using SEAs to associate information with files. They may define their own application-specific extended attributes. Applications define and associate extended attributes with a file object through file system function calls.

See the *OS/2 Version 2.0 Programming Guide* for a complete description of EA naming conventions and data types and standard extended attributes. See also the *OS/2 Version 2.0 Control Program Programming Reference* for a complete description of the file system function calls.

EAs may be viewed as a property list attached to file objects. The services for manipulating EAs are: add/replace a series of name/value pairs, return name/value pairs given a list of names, and return the total set of EAs.

There are two formats for EAs as passed to OS/2 Version 2.0 API: Full EAs (FEA) and Get EAs (GEA).

FEAs

FEAs are complete name/value pairs. In order to simplify and speed up scanning and processing of these names, they are represented as length-preceded data . FEAs are defined as follows:

```
struct FEA {
    unsigned char fEA;        /* byte of flags      */
    unsigned char cbName;     /* length of name     */
    unsigned short cbValue;   /* length of value    */
    unsigned char szName[];   /* ASCIIZ name        */
    unsigned char aValue[];   /* free format value  */
};
```

There is only one flag defined in fEA. That is 0x80 which is fNeedEA. Setting the flag marks this EA as needed for the proper operation on the file to which it is associated. Setting this bit has implications for access to this file by old applications, so it should not be set arbitrarily.

If a file has one or more NEED EAs, old applications are not allowed to open the file. For DOS mode applications to access files with NEED EAs, they must have the EA bit set in their exe header. For OS/2 mode, only applications with the NEWFILES bit set in the exe header may open files with NEED EAs.

Programs that change EAs should preserve the NEED bit in the EAs unless there is a good reason to change it.

The name length does not include the trailing NUL. The maximum EA name length is 255 bytes. The minimum EA name length is 1 byte. The characters that form the name are legal filename characters. Wildcard characters are not allowed . EA names are case-insensitive and should be uppercased. The FSD should call FSH_CHECKEANAME and FSH_UPPERCASE for each EA name it receives to check for invalid characters and correct length, and to uppercase it.

The FSD may not modify the flags.

A list of FEAs is a packed set of FEA structures preceded by a length of the list (including the length itself) as indicated in the following structure:

```
struct FEAList {
    unsigned long cbList;       /* length of list       */
    struct FEA list[];          /* packed set of FEAs   */
};
```

FEA lists are used for adding, deleting, or changing EAs. A particular FSD may store the EAs in whatever format it desires. Certain EAs may be stored to optimize retrieval.

GEAs

A GEA is an attribute name. Its format is:

```
struct GEA {
    unsigned char cbName;       /* length of name    */
    unsigned char szName[];     /* ASCIIZ name       */
};
```

The name length does not include the trailing NUL.

A list of GEAs is a packed set of GEA structures preceded by a length of the list (including the length itself) as indicated in the following structure:

```
struct GEAList {
    unsigned long cbList;     /* length of list     */
    struct GEA list[];        /* packed set of GEAs */
};
```

GEA lists are used for retrieving the values for a particular set of attributes. A GEA list is used as input only.

Name lengths of 0 are illegal and are considered in error. A value length of 0 has special meaning. Setting an EA with a value length of 0 will cause that attribute to be deleted (if possible). Upon retrieval, a value length of 0 indicates that the attribute is not present.

Setting attributes contained in an FEA list does not treat the entire FEA list as atomic. If an error occurs before the entire list of EAs has been set, all, some, or none of them may actually remain set on the file. No program should depend on an EA set being atomic to force EAs to be consistent with each other . Programs must be careful not to depend on atomicity, since a given file system may provide it.

Manipulation of extended attributes is associated with access permission to the associated file or directory. For querying and setting file EAs, read and write/read permission, respectively, for the associated file is required. No directory create or delete may occur while querying EAs for that directory.

For handle-based operations on extended attributes, access permission is controlled by the sharing/access mode of the associated file. If the file is open for read, querying the extended attributes is allowed. If the file is open for write , setting the extended attributes is allowed. These operations are DosQFileInfo and DosSetFileInfo.

For path-based manipulation of extended attributes, the associated file or directory will be added to the sharing set for the duration of the call. The requested access permission for setting EAs is write/deny-all and for querying EAs is read/deny-write. The path-based API are DosQPathInfo, DosSetPathInfo, and DosFindFirst2/DosFindNext.

For create-only operations of extended attributes, the extended attributes are set without examining the sharing/access mode of the associated file/directory . These operations are DosOpen2 and DosMkDir2.

The routing of EA requests is accomplished by the IFS routing mechanism. EA requests that apply to names are routed to the FSD attached to the specified drive. Those requests that apply to a handle (file or directory) are routed to the FSD attached to the handle. No interpretation of either FEA lists nor GEA lists is performed by the IFS router.

Note: It is the responsibility of each FSD to provide support for EAs .

It is expected that some FSDs will be unable to store EAs; for example, UNIX and MVS compatible file systems.

Note: The FAT FSD implementation will provide for the complete implementation of EAs. There will be no special EAs for the FAT FSD.

All EA manipulation is performed using the following structure: The relevance of each field is described within each API.

```
struct EAOP {
    struct GEAList far * fpGEAList; /* GEA set            */
    struct FEAList far * fpFEAList; /* FEA set            */
    unsigned long offError;         /* offset of FEA err */
};
```

See the descriptions of the file system function calls in *OS/2 Version 2. 0 Control Program Programming Reference* for the relevance of each field.

In OS/2 Version 2.0, values of cbList greater than (64K-1) are not allowed. This is an implementation-defined limitation which may be raised in the future. Because this limit may change, programs should avoid enumerating the list of all EAs, but instead manipulate only EAs that they know about. For operations such as copying, the DosCopy API should be used. If enumeration is necessary, the DosEnumAttribute API should be used.

A special category of attributes, called create-only attributes, is defined as the set of extended attributes that a file system may only allow to be set at creation time. (Such attributes may be used to control file allocation and structure configuration.) File systems are expected to allow create-only attributes to be set at any time from when the object is created to when it is first modified, that is, data is written into a file or an entry added to a directory. Programs that copy objects should copy all of the EAs for an object before otherwise modifying it in order to assure that any create-only attributes from the source are properly applied to the target. The DosCopy API is the preferred method of copying files or directories.

FSD File Image

An FSD loads from a file which is in the format of a standard OS/2 dynamic link library file. Exactly one FSD resides in each file. The FSD exports information to OS/2 using a set of predefined public names.

The FSD is initialized by a call to the exported entry point FS_INIT.

FS entry points for Mount, Read, Write, etc. are exported with known names as standard far entry points.

The FSD exports its name as a public ASCIIZ character string under the name 'FS_NAME'. All comparisons with user-specified strings are done similar to file names; case is ignored and embedded blanks are significant. FS_NAMEs, however, may be input to applications by users. Embedded blanks should be avoided . The name exported as FS_NAME need NOT be the same as the 1-8 FSD name in the boot sector of formatted media, although it may be. The ONLY name the kernel pays any attention to, and the only name accessible to user programs through the API , is the name exported as FS_NAME.

In addition to various entry points, the FSD must export a dword bit vector of attributes. Attributes are exported under the name 'FS_ATTRIBUTE' . FS_ATTRIBUTE specifies special properties of the FSD and is described in the next section.

FSD Attribute

The format of the OS/2 FS_ATTRIBUTE is defined in Figure 1-4 and the definition list that follows it.

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16

 E   V   V   V   R   R   R   R   R   R   R   R   R   R   R   R
 x   e   e   e   e   e   e   e   e   e   e   e   e   e   e   e
 A   r   r   r   s   s   s   s   s   s   s   s   s   s   s   s
 t   s   s   s   v   v   v   v   v   v   v   v   v   v   v   v

 R   R   R   R   R   R   R   R   R   R   R   R   L   F   U   R
 e   e   e   e   e   e   e   e   e   e   e   e   v   I   N   e
 s   s   s   s   s   s   s   s   s   s   s   s   l   /   C   m
 v   v   v   v   v   v   v   v   v   v   v   v   7   O       t

15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

Figure 1-4. OS/2 FSD Attribute

Bits Description

31 **FSD Additional attributes.** If 1, FSD has additional attributes. If 0, FS_ATTRIBUTE is the only FSD attribute information.

30-28 **VERSION NUMBER - FSD version number.**

27-4 **RESERVED**

3 **LEVEL7 - QPathInfo Level 7 bit.** Set if FSD is case-preserving. If this bit is set, the kernel will call the FS_PATHINFO entry point with a level equal to 7. The output buffer is to be filled with a case-preserved copy of the path that was passed in by the user.

2 **FILEIO - File I/O bit.** Set if FSD wants to see file locking/unlocking operations and compacted file I/O operations. If not set, the file I/O calls will be broken up into individual lock/unlock/read/write/seek calls and the FSD will not see the lock/unlock calls. FSDs that do not support file locking can set this bit to enable compacted file I/O operations.

1 **UNC - Universal Naming Convention bit.** Set if FSD supports the Universal Naming Convention. OS/2 Version 2.0 supports multiple loaded UNC redirectors.

0 **REMOTE - Remote File System (Redirector).** This bit tells the system whether the FSD uses static or dynamic media attachment. Local FSDs always use dynamic media attachment. Remote FSDs always use static media attachment. This bit is clear if it is a dynamic media attachment and set, if a static attachment . No FSD supports both static and dynamic media attachment. To support proper file locking, a remote FSD should also set the FILEIO bit.

FSD Initialization

FSD initialization occurs at system initialization time. FSDs are loaded through the IFS= configuration command in CONFIG.SYS. Once the FSD has been loaded, the FSD's initialization entry point is called to initialize it.

FSDs are structured the same as dynamic link library modules. Once an FSD is loaded, the initialization routine FS_INIT is called. This gives the FSD the ability to process any parameters that may appear on the CONFIG.SYS command line, which are passed as a parameter to the FS_INIT routine. A LIBINIT routine in an FSD will be ignored.

OS/2 FSDs initialize in protect mode. Because of the special state of the system, an FSD may make dynamic link system calls at init-time.

The list of systems calls that an FSD may make are as follows:

oDosBeep
oDosChgFilePtr
oDosClose
oDosDelete
oDosDevConfig
oDosDevIoCtl
oDosFindClose
oDosFindFirst
oDosFindNext
oDosGetEnv
oDosGetInfoSeg
oDosGetMessage
oDosOpen
oDosPutMessage
oDosQCurDir
oDosQCurDisk
oDosQFileInfo
oDosQFileMode
oDosQSysInfo
oDosRead
oDosWrite

The FSD may not call ANY FS helper routines at initialization time.

Note that multiple code and data segments are not discarded by the loader as in the case of device drivers.

The FSD may call DosGetInfoSeg to obtain access to the global and process local information segments. The local segment may be used in the context of all processes without further effort to make it accessible and has the same selector. The local infoseg is not valid in real mode or at interrupt time.

OS/2 and DOS Extended Boot Structure and BIOS Parameter Block

The Extended Boot structure is as follows:

```
struct Extended_Boot {
    unsigned char Boot_jmp[3];
    unsigned char Boot_OEM[8];
    struct Extended_BPB Boot_BPB;
    unsigned char Boot_DriveNumber;
    unsigned char Boot_CurrentHead;
    unsigned char Boot_Sig = 41; /* Indicate Extended Boot */
    unsigned char Boot_Serial[4];
    unsigned char Boot_Vol_Label[11];
    unsigned char Boot_System_ID[8];
};
```

Where

Boot_Serial is the 32-bit binary volume serial number for the media.

Boot_System_ID is an 8-byte name written when the media is formatted. It is used by FSDs to identify their media but need not be the same as the name the FSD exports via FS_NAME and is NOT the name users employ to refer to the FSD. (They may, however, be the same names).

Boot_Vol_Label is the 11-byte ASCII label of the disk/diskette volume. FAT file systems must ALWAYS use the volume label in the root directory for compatibility reasons. An FSD may use the one in the boot sector.

The extended BPB structure is a super-set of the conventional BPB structure, as follows:

```
struct Extended_BPB {
    unsigned short BytePerSector;
    unsigned char SectorPerCluster;
    unsigned short ReservedSectors;
    unsigned char NumberOfFats;
    unsigned short RootEntries;
    unsigned short TotalSectors;
    unsigned char MediaDescriptor;
    unsigned short SectorsPerFat;
    unsigned short SectorsPerTrack;
    unsigned short Heads;
    unsigned long HiddenSectors;
    unsigned long Ext_TotalSectors;
};
```

IFS Commands

IFS = (CONFIG.SYS Command)

An FSD is loaded and initialized at system start-up when an IFS= statement is encountered in CONFIG.SYS. The syntax of this command is as follows:

```
IFS=drive:path\name.ext parms
```

where
drive:path\name.ext specifies the FSD to load and initialize.
parms represents an FSD-defined string of initialization parameters.

See the *OS/2 Version 2.0 Online Command Reference* for a detailed description of this command.

File System Function Calls

The *OS/2 Version 2.0 Control Program Programming Reference* gives a detailed description of the 32-bit file system calls new for OS/2 Version 2.0 See the *OS/2 Version 2.0 Programming Guide* for a description of how to use these calls. For detailed descriptions of the 16-bit file system calls see the *OS/2 Version 1.3 Control Program Programming Reference* , and the *OS/2 Version 1.3 Programming Guide* on how to use these calls.

Note: The data structures for some of the file system calls have changed in their 32-bit implementations. For OS/2 Version 2.0 the kernel will handle all remapping between the 32-bit structures and the 16-bit structures used by individual FSDs.

Application File I/O Notes

File handle values of 0xFFFF do not represent actual file handles but are used throughout the file system interface to indicate specific actions to be taken by the file system. Usage of this special file handle where it is not expected by the file system will result in an error.

Null pointers are defined to be 0x00000000 throughout this document.

File systems that conform to the Standard Application Program Interface (Standard API) may not necessarily support all the described information kept on a file basis. When this is the case, FSDs are required to return to the application a null (zero) value for the unsupported parameter.

An FSD may support version levels of files.

Date/Time Stamps

The format of OS/2 dates are show below in Figure 1-5.

```
 Y   Y   Y   Y   Y   Y   Y   M   M   M   M   D   D   D   D   D
 e   e   e   e   e   e   e   o   o   o   o   a   a   a   a   a
 a   a   a   a   a   a   a   n   n   n   n   y   y   y   y   y
 r   r   r   r   r   r   r   t   t   t   t
                             h   h   h   h

15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

Figure 1-5. OS/2 Date Format

Bits Description
15-9 YEARS - Number of years since 1980.
8-5 MONTH - is the month of the year (1-12)
4-0 DAY - is the day of the month (1-31)

The format of OS/2 times are show below in Figure 1-6.

```
 H   H   H   H   H   M   M   M   M   M   M   2   2   2   2   2
 o   o   o   o   o   i   i   i   i   i   i
 u   u   u   u   u   n   n   n   n   n   n   S   S   S   S   S
 r   r   r   r   r                           e   e   e   e   e
                                             c   c   c   c   c

15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

Figure 1-6. OS/2 Time Format

Bits Description
15-9 HOUR - is the hour of the day (0-23)
8-5 MINUTE - is the minute of the hour (0-59)
4-0 2-SECOND - is the second of the minute(in increments of 2) (0-29)

I/O Error Codes

Some file system functions may return device-driver/device-manager generated errors. These include:

oERROR_WRITE_PROTECT - the media in the drive has write-protection enabled .
oERROR_BAD_UNIT - there is a breakdown of internal consistency between OS/2 's mapping between logical drive and device driver. Internal Error.
oERROR_NOT_READY - the device driver detected that the device is not ready.
oERROR_BAD_COMMAND - there is a breakdown of internal consistency between OS/2 's idea of the capability of a device driver and that of the device driver.
oERROR_CRC - the device driver has detected a CRC error.
oERROR_BAD_LENGTH - there is a breakdown of internal consistency between OS/2 's idea of the length of a request packet and the device driver's idea of that length. Internal Error.
oERROR_SEEK - the device driver detected an error during a seek operation.
oERROR_NOT_DOS_DISK - the disk is not recognized as being OS/2 manageable.
oERROR_SECTOR_NOT_FOUND - the device is unable to find the specific sector.
oERROR_OUT_OF_PAPER - the device driver has detected that the printer is out of paper.
oERROR_WRITE_FAULT - other write-specific error.
oERROR_READ_FAULT - other read-specific error.
oERROR_GEN_FAILURE - other error.

There are also errors defined by and specific to the device drivers. These are indicated by either 0xFF or 0xFE in the high byte of the error code.

Note: Error codes listed in the function call descriptions in the *OS/2 Version 2.0 Control Program Programming Reference* are not complete. They are errors most likely to be returned by the FS router and the FAT file system. Each FSD may generate errors based upon its own circumstances.

FSD System Interfaces

Overview

Installable file system entry points are called by the kernel as a result of action taken through the published standard file I/O application programming interface in OS/2 Version 2.0.

Installable file systems are installed as OS/2 dynamic link library modules . Unlike device drivers, they may include any number of segments, all of which will remain after initialization, unless the FSD itself takes some action to free them.

An FSD exports FS entries to the OS/2 kernel using standard PUBLIC declarations. Each FS entry is called directly. The OS/2 kernel manages the association between internal data structures and FSDs.

When a file system service is required, OS/2 assembles an argument list, and calls the appropriate FS entry for the relevant FSD. If a back-level FSD is loaded, the OS/2 kernel assures that all arguments passed and all structures passed are understood by the FSD.

Application program interfaces that are unsupported by an FSD receive an UNSUPPORTED FUNCTION error from the FSD.

Certain routines, for example, FS_PROCESSNAME, may provide no processing, no processing is needed, or processing does not make sense. These routines return no error, not ERROR_NOT_SUPPORTED.

Data Structures

OS/2 data structures that include a pointer to the file system driver, as well as file system specific data areas are:

othe CDS (current directory structure)
othe SFT (system file table entry),
othe VPB (volume parameter block)
othe file search structures.

File system service routines are generally passed pointers to two parameter areas, in addition to read-only parameters which are specific to each call. The FSD does not need to verify these pointers. The two parameter areas contain file-system-independent data which is maintained jointly by OS/2 and the file system driver and file-system-dependent data which is unused by OS/2 and which may be used in any way by the file system driver. The file system driver is generally permitted to use the file-system-dependent information in any way. The file-system-dependent information may contain all the information needed to describe the current state of the file or directory, or it may contain a handle which will direct it to other information about the file maintained within the FSD. Handles must be GDT selectors because any SFT, CDS, or VPB may be seen by more than one process. File- system-dependent and file-system-independent parameter areas are defined by data structures described in the remainder of this section.

Disk media and file system layout

are described by the following structures. The data which is provided to the file system may depend on the level of file system support provided by the device driver attached to the block device. These structures are relevant only for local file systems.

```
/* file system independent - volume parameters */

struct vpfsi {
    unsigned long  vpi_vid;         /* 32 bit volume ID */
    unsigned long  vpi_hDEV;        /* handle to device driver */
    unsigned short vpi_bsize;       /* sector size in bytes */
    unsigned long  vpi_totsec;      /* total number of sectors */
    unsigned short vpi_trksec;      /* sectors / track */
    unsigned short vpi_nhead;       /* number of heads */
    char           vpi_text[12];    /* ASCIIZ volume name */
    void far *     vpi_pDCS;        /* device capability structure */
    void far *     vpi_pVCS;        /* volume characteristics */
    unsigned char  vpi_drive;       /* drive (0=A) */
    unsigned char  vpi_unit;        /* unit code */
};

/* file system dependent - volume parameters */

struct vpfsd {
    char           vpd_work[36];   /* work area */
};
```

Per-disk current directories

are described by the following structures. These structures can only be modified by the FSD during FS_ATTACH and FS_CHDIR operations.

```
/* file system independent - current directories */

struct cdfsi {
    unsigned short cdi_hVPB;           /* VPB handle for associated device */
    unsigned short cdi_end;            /* offset to root of path */
    char           cdi_flags;          /* FS independent flags */
    char           cdi_curdir[260];    /* text of current directory */
};

/* file system dependent - current directories */

struct cdfsd {
    char           cdd_work[8];        /* work area */
};
```

Open files

are described by data initialized at file open time and discarded at the time of last close of all file handles which had been associated with that open instance of that file. There may be multiple open file references to the same file at any one time.

All time stamps on files are stamped and propagated to other SFTs by OS/2 when the file is closed or committed (flushed). For example, if a file is opened at time 1, written at time 2, and closed at time 3, the last write time is time 3 . Subdirectories need only have creation time stamps because the last write and last read time stamps on subdirectories are either very difficult to implement (propagate up to parent subdirectories), or are not very useful. An FSD, however, may implement them. FSDs are required to support direct access opens. These are indicated by a bit set in the sffsi.sfi_mode field.

```
/* file system independent - file instance */

struct sffsi {
    unsigned long   sfi_mode;       /* access/sharing mode */
    unsigned short  sfi_hVPB;       /* volume info. */
    unsigned short  sfi_ctime;      /* file creation time */
    unsigned short  sfi_cdate;      /* file creation date */
    unsigned short  sfi_atime;      /* file access time */
    unsigned short  sfi_adate;      /* file access date */
    unsigned short  sfi_mtime;      /* file modification time */
    unsigned short  sfi_mdate;      /* file modification date */
    unsigned long   sfi_size;       /* size of file */
    unsigned long   sfi_position;   /* read/write pointer */

/* the following may be of use in sharing checks */

    unsigned short  sfi_UID;        /* user ID of initial opener */
    unsigned short  sfi_PID;        /* process ID of initial opener */
    unsigned short  sfi_PDB;        /* PDB (in 3.x box) of initial opener */
    unsigned short  sfi_selfsfn;    /* system file number of file instance */
    unsigned char   sfi_tstamp;     /* time stamp flags */
    unsigned short  sfi_type;       /* type of object opened */
    unsigned long   sfi_pPVDBFil;   /* performance counter data block pointer */
    unsigned char   sfi_DOSattr;    /* DOS file attributes D/S/A/H/R */
};

/* file system dependent - file instance */
```

```
struct sffsd {
    char            sfd_work[30];   /* work area */
};
```

The Program Data Block, or PDB (sfi_pdb), is the unit of sharing for DOS mode processes. For OS/2 mode processes, the unit of sharing is the Process ID, PID (sfi_ pid). FSDs should use the combination PDB, PID, UID as indicating a distinct process.

File search records

```
/* file system independent - file search parameters */

struct fsfsi {
        unsigned short  fsi_hVPB;       /* volume info. */
};

/* file system dependent - file search parameters */

struct fsfsd {
        char            fsd_work[24];   /* work area */
};
```

Existing file systems that conform to the Standard Application Program Interface (Standard API) described in this section, may not necessarily support all the described information kept on a file basis. When this is the case, file system drivers are required to return to the application a null (zero) value for the unsupported parameter (when the unsupported data are a subset of the data returned by the API) or to return a ERROR_NOT_SUPPORTED error (when all of the data returned by the API is unsupported).

Time Stamping

All time stamps on files are stamped and propagated to other SFTs when the file is closed or committed (flushed). If a file is opened at time 1, written to at time 2, and closed at time 3, the last write time will be time 3. Subdirectories only have creation time stamps.

The sfi_tstamp field of the file instance structure sffsi contains six flags :

Name	Value	Description
ST_SCREAT	1	stamp creation time
ST_PCREAT	2	propagate creation time
ST_SWRITE	4	stamp last write time
ST_PWRITE	8	propagate last write time
ST_SREAD	16	stamp last read time
ST_PREAD	32	propagate last read time

These flags are cleared when an SFT is created, and some of them may eventually be set by a file system worker routine. They are examined when the file is closed or flushed.

For each time stamp, there are three meaningful actions:

ST_Sxxx	ST_Pxxx	Action

clear	clear	don't do anything
set	set	stamp and propagate (to other SFTs and disk)
clear	set	don't stamp, but propagate existing value

FSD Calling Conventions and Requirements

Calling conventions between FS router, FSD, and FS helpers are:

oArguments will be pushed in left-to-right order onto the stack.

oThe callee is responsible for cleaning up the stack.

oRegisters DS, SI, DI, BP, SS, SP are preserved.

oReturn conditions appear in AX with the convention that AX == 0 indicates successful completion. AX != 0 indicates an error with the value of AX being the error code.

Interrupts must ALWAYS be enabled and the direction flag should be presumed to be undefined. Calls to the FS helpers will change the direction flag at will.

In OS/2, file system drivers are always called in kernel protect mode. This has the advantage of allowing the FSD to execute code without having to account for preemption; no preemption occurs when in kernel mode. While this greatly simplifies FSD structure, it forces the FSD to yield the CPU when executing long segments of code. In particular, an FSD must not hold the CPU for more than 2 milliseconds at a time. The FSD helper FSH_YIELD is provided so that FSDs may relinquish the CPU.

File system drivers cannot have any interrupt-time activations. Because they occupy high, movable, and swappable memory, there is no guarantee of addressability of the memory at interrupt time.

Each FS service routine may block.

Error Codes

FSDs should use existing error codes when possible. New error codes must be in the range reserved for FSDs. The FS_FSCTL interface must support returning information about new error codes.

The set of error codes for errors general to all FSDs is 0xEE00 - 0xEEFF. The following errors have been defined:

oERROR_VOLUME_NOT_MOUNTED = 0xEE00 - the FSD did not recognize the volume .

The set of error codes which are defined by each FSD is 0xEF00 - 0xFEFF.

FS Service Routines

The following table summarizes the entry points that make up the interface between the kernel and the FSD.

Note: Names must be in all upper case, as required by OS/2 naming conventions.

FS Entry Point	Description	FSDs Required to export
FS_ALLOCATEPAGESPACE	Adjust the size of paging file	PAGE I/O

FS_ATTACH	Attach to an FSD	ALL
FS_CANCELLOCKREQUEST	Cancel file record lock request	FILE I/O
FS_CHDIR	Change/Verify directory path	ALL
FS_CHGFILEPTR	Move a file's position pointer	ALL
FS_CLOSE	Release a file handle	ALL
FS_COMMIT	Flush a file's buffer to disk	ALL
FS_COPY	Copy a file	ALL
FS_DELETE	Delete a file	ALL
FS_DOPAGEIO	Perform paging I/O operations	PAGE I/O
FS_EXIT	End of a process cleanup	ALL
FS_FILEATTRIBUTE	Query/Set file's attributes	ALL
FS_FILEINFO	Query/Set file's information	ALL
FS_FILEIO	Multi-function file I/O	ALL
FS_FILELOCKS	Request a file record lock/unlock	FILE I/O
FS_FINDCLOSE	Directory search close	ALL
FS_FINDFIRST	Find first matching filename	ALL
FS_FINDFROMNAME	Find matching filename from name	ALL
FS_FINDNEXT	Find next matching filename	ALL
FS_FINDNOTIFYCLOSE	Close FindNotify handle	ALL
FS_FINDNOTIFYFIRST	Monitor a directory for changes	ALL
FS_FINDNOTIFYNEXT	Resume reporting directory changes	ALL
FS_FLUSHBUF	Commit file buffers to disk	ALL
FS_FSCTL	File system control	ALL
FS_FSINFO	Query/Set file system information	ALL
FS_INIT	FSD initialization	ALL
FS_IOCTL	I/O device control	ALL
FS_MKDIR	Make a directory	ALL
FS_MOUNT	Mount/unmount volumes	ALL
FS_MOVE	Move a file or subdirectory	ALL
FS_NEWSIZE	Change a file's logical size	ALL
FS_NMPIPE	Do a named pipe operation	ALL
FS_OPENCREATE	Open/create/replace files	ALL
FS_OPENPAGEFILE	Create paging file and handle	PAGE I/O
FS_PATHINFO	Query/Set a file's information	ALL
FS_PROCESSNAME	FSD unique name canonicalization	ALL
FS_READ	Read data from a file	ALL
FS_RMDIR	Remove a subdirectory	ALL
FS_SETSWAP	Notification of swapfile ownership	ALL
FS_SHUTDOWN	Shutdown file system	ALL
FS_VERIFYUNCNAME	Verify UNC server ownership	UNC
FS_WRITE	Write data to a file	ALL

Each FS entry point has a distinct parameter list composed of those parameters needed by that particular entry. Parameters include:

oFile pathname
oCurrent disk/directory information
oOpen file information
oApplication data buffers
oDescriptions of file extended attributes
oOther parameters specific to an individual call

Most of the FS entry points have a level parameter for specifying the level of information they are provided or have to supply. FSDs must provide for additional levels which may be added in future versions of OS/2 by returning ERROR_NOT_ SUPPORTED for any level they do not recognize.

File system drivers which support hierarchical directory structures must use '\' and '/' as path name component separators. File system drivers which do not support hierarchical directory structures must reject as illegal any use of '\' or '/' in path names. The file names '.' and '..' are reserved for use in hierarchical directory structures for the current directory and the parent of the current directory, respectively.

Unless otherwise specified in the descriptions below, data buffers may be accessed without concern for the accessibility of the data. OS/2 will either check buffers for accessibility and lock them, or transfer them into locally accessible data areas.

Simple parameters will be verified by the IFS router before the FS service routine is called.

Note: New with 2.0, some entry points need only be exported and supported by those FSDs which desire to service the pager (PAGE I/0), UNC servers (UNC) and/or file locking (FILE I/O). With these new entry point groups, a FSD must export all or none of the entry points within a particular group.

These optional entry points are:

```
FS_ALLOCATEPAGESPACE     (PAGE I/O)
FS_CANCELLOCKREQUEST     (FILE I/O)
FS_DOPAGEIO              (PAGE I/O)
FS_FILELOCKS             (FILE I/O)
FS_OPENPAGEFILE          (PAGE I/O)
FS_VERIFYUNCNAME         (UNC)
```

FS_ALLOCATEPAGESPACE - Adjust the size of paging file

Purpose

Changes the size the paging file on disk.

Calling Sequence

```
int far pascal FS_ALLOCATEPAGESPACE(psffsi, psffsd, ulsize, ulWantContig)

struct sffsi far * psffsi;
struct sffsd far * psffsd;
unsigned long ulsize;
unsigned short ulWantContig;
```

Where

psffsi is a pointer to the file-system-independent portion of an open file instance.

psffsd is a pointer to the file-system-dependent portion of an open file instance.

ulsize is the desired new size of the paging file. If the new size is smaller than the current size, the excess space is released. If the new size is larger than the current size, the requested size is allocated.

ulWantContig indicates the minimum contiguity requirement (in bytes).

Remarks

ulWantContig is a demand for contiguity. If ulWantContig is non-zero(0), the FSD must allocate any space in the swap file that is not contiguous in ulWantContig chunks on ulWantContig boundaries. If it is not possible to grow the file to ulSize bytes meeting the ulWantContig requirement, the operation should fail . If the file is being shrunk ulWantContig is irrelevant and should be ignored .

FSDs that support the paging I/O interface should be expected to be sensible in allocating page space. In particular, they are expected to always attempt to allocate space such that ulWantContig sized blocks on ulWantContig boundaries are physically contiguous on disk, and to keep the page file as a whole contiguous as possible.

FS_ATTACH - Attach to an FSD

Purpose

Attach or detach a remote drive or pseudo-device to an FSD.

Calling Sequence

```
int far pascal FS_ATTACH(flag, pDev, pvpfsd, pcdfsd, pParm, pLen)

unsigned short flag;
char far * pDev;
struct vpfsd far * pvpfsd;
struct cdfsd far * pcdfsd;
char far * pParm;
unsigned short far * pLen;
```

Where

flag indicates attach or detach:

flag == 0 requests an attach. The FSD is being called to attach a specified driver or character device.
flag == 1 requests a detach.
flag == 2 requests the FSD to fill in the specified buffer with attachment information.

pDev is a pointer to the ASCIIZ text of either the driver (driver letter followed by a colon) or to the character device (must be \DEV\device) that is being attached, detached, or queried. The FSD does not need to verify this pointer.

tpvpfsd is a pointer to a data structure containing file-system-dependent volume parameter information. When an attach/detach/query of a character device is requested, this pointer is NULL. When attaching a drive, this structure contains no data and is available for the FSD to store information needed to manage the remote drive. All subsequent FSD calls have access to the hVPB in one of the structures passed in, so the FSD has access to this structure by using FSH_GETVOLPARMS. This structure will have its contents as the FSD had left them. When detaching or querying a drive, this structure contains the data as the FSD left them.

pcdfsd is a pointer to a data structure containing file-system dependent working directory information for drives. When attaching a drive, this structure contains no data and is available for the FSD to store information needed to manage the working directory. All subsequent FSD calls generated by API calls that reference this drive are passed a pointer to this structure with contents left as the FSD left them. When detaching or querying a drive, this structure contains the data as the FSD left them. For character devices, pcdfsd points to a DWORD. When a device is attached, the DWORD contains no data, and can be used by the FSD to store a reference to identify the device later on during FS_OPENCREATE, when it is passed in to the FSD. When detaching or querying the device, this DWORD contains the data as the FSD left them.

pParm is the address of the application parameter area.

When an attach is requested, this will point to the API-specified user data block that contains information regarding the attach operation (for example, passwords). For a query, the OS/2 kernel will fill in part of the buffer, adjust the pointer, and call the FSD to fill in the rest (see the structures returned by DosQFSAttach; pParm will point to cbFSAData; the FSD should fill in cbFSAData and rgFSAData .)

Addressing of this data area is not validated by the OS/2 kernel. pParm must be verified, even in the query case. The FSD verifies this parameter by calling the FS helper routine FSH_PROBEBUF.

pLen is the pointer to the length of the application parameter area. For attach, this points to the length of the application data buffer. For query, this is the length of the remaining space in the application data buffer. Upon filling in the buffer, the FSD will set this to the length of the data returned. If the data returned is longer than the data buffer length, the FSD sets this value to be the length of the data that query could return. In this case, the FSD also returns a BUFFER OVERFLOW error.

The FSD does not need to verify this pointer.

Remarks

Local FSDs will never get called with attempts to attach or detach drives or queries about drives.

For remote FSDs called to do a detach, the kernel does not do any checking to see if there are any open references on the drive (for example, open or search references). It is entirely up to the FSD to decide whether it should allow the detach operation.

FS_CANCELLOCKREQUEST - Cancel file record lock request

Purpose

Cancels an outstanding FS_FILELOCKS request on a file.

Calling Sequence

```
int far pascal FS_CANCELLOCKREQUEST(psffsi, psffsd, pLockRange)

struct sffsi far * psffsi;
struct sffsd far * psffsd;
struct filelock far * pLockRange;
```

Where

psffsi is a pointer to the file-system-independent portion of an open file instance.

psffsd is a pointer to the file-system-dependent portion of an open file instance.

pLockRange is a pointer to a filelock structure. The filelock structure has the following format:

```
struct FileLock {
    unsigned long FileOffset;   /* offset where the lock/unlock begins  */
    unsigned long RangeLength;  /* length of region locked/unlocked     */
}
```

Remarks

This entry point was added to support the 32-bit DosCancelLockRequest API .

This function provides a simple mechanism for canceling the lock range request of an outstanding FS_FILELOCKS call. If two threads in a process are blocked on a lock range and a cancel request is issued by another thread, both blocked threads will be released.

FS_CHDIR - Change/Verify Directory Path

Purpose

Change or verify the directory path for the requesting process

Calling Sequence

```
int far pascal FS_CHDIR(flag, pcdfsi, pcdfsd, pDir, iCurDirEnd)

unsigned short flag;
struct cdsfi far * pcdfsi;
struct cdfsd far * pcdfsd;
char far * pDir;
unsigned short iCurDirEnd;
```

Where

flag indicates what action is to be taken on the directory.

flag == 0 indicates that an explicit directory-change request has been made .
flag == 1 indicates that the working directory needs to be verified.
flag == 2 indicates that this reference to a directory is being freed.

The flag passed to the FSD will have a valid value.

pcdfsi is a pointer to a file-system-independent working directory structure .

For flag == 0, this pointer points to the previous current directory on the drive.
For flag == 1, this pointer points to the most recent working directory on the drive. The cdi_curdir field contains the text of the directory that is to be verified.
For flag == 2, this pointer is null.

The FSD must never modify the cdfsi. The OS/2 kernel handles all updates .

pcdfsd is a pointer to a file-system-dependent working directory structure .

This is a place for the FSD to store information about the working directory . For flag == 0 or 1, this is the information left there by the FSD. The FSD is expected to update this information if the directory exists. For flag == 2, this is the information left there by the FSD.

pDir is a pointer to directory text.

For flag == 0, this is the pointer to the directory. For flag == 1 or flag == 2, this pointer is null. The FSD does not need to verify this pointer .

iCurDirEnd is the index of the end of the current directory in pDir.

This is used to optimize FSD path processing. If iCurDirEnd == -1, there is no current directory relevant to the directory text, that is, a device. This parameter only has meaning for flag == 0.

Remarks

The FSD should cache no information when the directory is the root. Root directories are a special case. They always exist, and never need validation. The OS/2 kernel does not pass root directory requests to the FSD. An FSD is not allowed to cache any information in the cdfsd data structure for a root directory. Under normal conditions, the kernel does not save the CDS for a root directory and builds one from scratch when it is needed. (One exception is where a validate CDS fails, and the kernel sets it to the root, and zeroes out the cdfsd data structure . This CDS is saved and is cleaned up later.)

The following is information about the exact state of the cdfsi and cdfsd data structures passed to the FSD for each flag value and guidelines about what an FSD should do upon receiving an FS_CHDIR call:

```
IF (flag == 0)      /* Set new Current Directory */
   pcdfsi, pcdfsd = copy of CDS we're starting from; may be useful as starting
                    point for verification.

           cdfsi contents:

              hVPB - handle of Volume Parameter Block mapped to this drive

              end - end of root portion of CurDir

              flags - various flags (indicating state of cdfsd)

              IsValid - cdfsd is unknown format (ignore contents)
                        IsValid == 0x80

              IsRoot - cdfsd is meaningless if CurDir = root (not kept)
                        IsRoot == 0x40

              IsCurrent - cdfsd is know format, but may not be current (medium
                          may have been changed).
                          IsCurrent == 0x20

              text - Current Directory Text

   icurdir = if Current Directory is in the path of the new Current Directory,
             this is the index to the end of the Current Directory. If not,
             this is -1 (Current Directory does not apply).

   pDir = path to verify as legal directory
```

```
THEN

   Validate path named in pDir.
       /* This means both that it exists AND that it is a directory.  pcdfsi,
          pcdfsd, icurdir give old CDS, which may allow optimization */

   IF (Validate succeeds)
      IF (pDir != ROOT)
         Store any cache information in area pointed to by pcdfsd.
      ELSE
         Do Nothing.
         /* Area pointed to by pcdfsd will be thrown away, so don't bother
            storing into it */
         Return success.
    ELSE
      Return failure.
      /* Kernel will create new CDS using pDir data and pcdfsd data. If the
         old CDS is valid, the kernel will take care of cleaning it up. The
         FSD must not edit any structure other than the *Pcdfsd area, with
         which it may do as it chooses. */
/* flag == 0 */
ELSE
IF (flag == 1)       /* Validate current CDS structure */

   pcdfsi = pointer to copy of cdfsi of interest.

   pcdfsd = pointer to copy of cdfsd. Flags in cdfsi indicate the state of
            this cdfsd. It may be: (1) completely invalid (unknown
            format), (2) known format, but non-current information,
            (3) completely valid, or (4) all zero (root).

THEN

    Validate that CDS still describes a legal directory (using cdi_text).

    IF (valid)
       Update cdfsd if necessary.
       Return success.
       /* kernel will copy cdfsd into real CDS */
    ELSE
       IF (cdi_isvalid)
          Release any resources associated with cdfsd.
          /* kernel will force Current Directory to root, and will zero out
             cdfsd in real CDS */
          Return failure.
    /* The FSD must not modify any structure other than the cdfsd pointed to by
       pcdfsd.  */
ELSE
IF (flag == 2)   /* previous CDS no longer in use; being freed */

   pcdfsd = pointer to copy of cdfsd of CDS being freed.

THEN

   Release any resources associated with the CDS.
   /* For example, if cdfsd (where pcdfsd points) contains a pointer to
      some FSD private structure associated with the CDS, that structure
      should be freed. */

/* kernel will not retain the cdfsd */
```

FS_CHGFILEPTR - Move a file's position pointer

Purpose

Move a file's logical read/write position pointer.

Calling Sequence

```
int far pascal FS_CHGFILEPTR(psffsi, psffsd, offset, type, IOflag)
```

```
struct sffsi far * psffsi;
struct sffsd far * psffsd;
long offset;
unsigned short type;
unsigned short IOflag;
```

Where

psffsi is a pointer to the file-system-independent portion of an open file instance.

The FSD uses the current file size or sfi_position along with offset and type to compute a new sfi_position. This is updated by the system.

psffsd is a pointer to the file-system-dependent portion of an open file instance. The FSD may store or adjust data as appropriate in this structure.

offset is the signed offset to be added to the current file size or position to form the new position within the file.

type indicates the basis of a seek operation.

type == 0 indicates seek relative to beginning of file.
type == 1 indicates seek relative to current position within the file.
type == 2 indicates seek relative to end of file.

The value of type passed to the FSD will be valid.

IOflag indicates information about the operation on the handle.

IOflag == 0x0010 indicates write-through.
IOflag == 0x0020 indicates no-cache.

Remarks

The file system may want to take the seek operation as a hint that an I/O operation is about to take place at the new position and initiate a positioning operation on sequential access media or read-ahead operation on other media.

Some DOS mode programs expect to be able to do a negative seek. OS/2 passes these requests on to the FSD and returns an error for OS/2 mode negative seek requests. Because a seek to a negative position is, effectively, a seek to a very large offset, it is suggested that the FSD return end-of-file for subsequent read requests.

FSDs must allow seeks to positions beyond end-of-file.

The information passed in IOflag is what was set for the handle during a DosOpen/DosOpen2 operation, or by a DosSetFHandState call.

FS_CLOSE - Close a file.

Purpose

Closes the specified file handle.

Calling Sequence

```
int far pascal FS_CLOSE(type, IOflag, psffsi, psffsd)

unsigned short type;
unsigned short IOflag;
struct sffsi far * psffsi;
struct sffsd far * psffsd;
```

Where

type indicates what type of a close operation this is.

type == 0 indicates that this is not the final close of the file or device .
type == 1 indicates that this is the final close of this file or device for this process.
type == 2 indicates that this is the final close for this file or device for the system.

IOflag indicates information about the operation on the handle.

IOflag == 0x0010 indicates write-through.
IOflag == 0x0020 indicates no-cache.

psffsi is a pointer to the file-system-independent portion of an open file instance.

psffsd is a pointer to the file-system-dependent portion of an open file instance.

Remarks

This entry point is called on the every close of a file or device.

Any reserved resources for this instance of the open file may be released It may be assumed that all open files will be closed at process termination. That is, this entry point will always be called at process termination for any files or devices open for the process.

A close operation should be interpreted by the FSD as meaning that the file should be committed to disk as appropriate.

Of the information passed in IOflag, the write-through bit is a mandatory bit in that any data written to the block device must be put out on the medium before the device driver returns. The no-cache bit, on the other hand, is an advisory bit that says whether the data being transferred is worth caching or not.

FS_COMMIT - Commit a file's buffers to Disk

Purpose

Flush requesting process's cache buffers and update directory information for the file handle.

Calling Sequence

```
int far pascal FS_COMMIT(type, IOflag, psffsi, psffsd)

unsigned short type;
unsigned short IOflag;
struct sffsi far * psffsi;
struct sffsd far* psffsd;
```

Where

type indicates what type of a commit operation this is.

type == 1 indicates that this is a commit for a specific handle. This type is specified if FS_COMMIT is called for a DosBufReset of a specific handle.

type == 2 indicates that this is a commit due to a DosBufReset (-1).

IOflag indicates information about the operation on the handle.

IOflag == 0x0010 indicates write-through.
IOflag == 0x0020 indicates no-cache.

psffsi is a pointer to the file-system-independent portion of an open file instance.

psffsd is a pointer to the file-system-dependent portion of an open file instance.

Remarks

This entry point is called only as a result of a DosBufReset function call . OS/2 reserves the right to call FS_COMMIT even if no changes have been made to the file.

For DosBufReset (-1), FS_COMMIT will be called for each open handle on the FSD.

The FSD should update access and modification times, if appropriate.

Any locally cached information about the file must be output to the media . The directory entry for the file is to be updated from the sffsi and sffsd data structures.

Since mini-FSDs used to boot IFSs are read-only file systems, they need not support the FS_COMMIT call.

Of the information passed in IOflag, the write-through bit is a MANDATORY bit in that any data written to the block device must be put out on the medium before the device driver returns. The no-cache bit, on the other hand, is an advisory bit that says whether the data being transferred is worth caching or not.

The FSD should copy all supported time stamps from the SFT to the disk. Beware that the last read time stamp may need to be written to the disk even though the file is clean. After this is done, the FSD should clear the sfi_tstamp field to avoid having to write to the disk again if the user calls commit repeatedly without changing any of the time stamps.

If the disk is not writeable and only the last read time stamp has changed, the FSD should either issue a warning or ignore the error. This relieves the user from having to un-protect an FSD floppy disk in order to read the files on it .

FS_COPY - Copy a file

Purpose

Copy a specified file or subdirectory to a specified target.

Calling Sequence

```
int far pascal FS_COPY(flag, pcdfsi, pcdfsd, pSrc, iSrcCurDirEnd, pDst,
                       iDstCurDirEnd, nameType)

unsigned short flag;
struct cdfsi far * pcdfsi;
struct cdfsd far * pcdfsd;
char far * pSrc;
unsigned short iSrcCurDirEnd;
char far * pDst; unsigned short iDstCurDirEnd;
unsigned short nameType;
```

Where

flag is a bit mask controlling copy

0x0001 specifies that an existing target file/directory should be replaced
0x0002 specifies that a source file will be appended to the destination file .
All other bits are reserved.

(See the description of the DosCopy function call in the *OS/2 Version 2.0 Control Program Programming Reference* .)

pcdfsi is a pointer to the file-system-independent working directory structure.

pcdfsd is a pointer to the file-system-dependent working directory structure .

pSrc is a pointer to the ASCIIZ name of the source file/directory.

iSrcCurDirEnd is the index of the end of the current directory in pSrc. If = -1, there is no current directory relevant to the source name.

pDst is a pointer to the ASCIIZ name of the destination file/directory.

iDstCurDirEnd is the index of the end of the current directory in pDst. If = -1, there is no current directory relevant to the destination name.

nameType indicates the destination name type.

NameType == 0x0040 indicates non-8.3 filename format. All other values are reserved.

Remarks

The file specified in the source file name should be copied to the target file if possible.

The files specified may not be currently open. File system drivers must assure consistency of file allocation information and directory entries.

The file system driver returns the special CANNOT COPY error if it cannot perform the copy because:

oit does not know how
othe source and target are on different volumes
oof any other reason for which it would make sense for its caller to perform the copy operation manually.

Returning ERROR_CANNOT_COPY indicates to its caller that it should attempt to perform the copy operation manually. Any other error will be returned directly to the caller of DosCopy. See the description of the DosCopy function call in the *OS/2 Version 2.0 Control Program Programming Reference* for other error codes that can be returned.

FS_COPY needs to check that certain types of illegal copying operations are not performed. A directory cannot be copied to itself or to one of its subdirectories. This is especially critical in situations where two different fully- qualified pathnames can refer to the same file or directory. For example, if X: is redirected to \\SERVER\SHARE, the X:\PATH and \\ SERVER\SHARE\PATH refer to the same object.

The behavior of FS_COPY should match the behavior of the generic DosCopy routine.

The non-8.3 filename format attribute in the directory entry for the destination name should be set according to the value in nameType.

FS_DELETE - Delete a file

Purpose

Removes a directory entry associated with a filename.

Calling Sequence

```
int far pascal FS_DELETE(pcdfsi, pcdfsd, pFile, iCurDirEnd)

struct cdfsi far * pcdfsi;
struct cdfsd far * pcdfsd;
char far * pFile;
unsigned short iCurDirEnd;
```

Where

pcdfsi is a pointer to the file-system-independent working directory structure.

pcdfsd is a pointer to the file-system-dependent working directory structure .

pFile is a pointer to the ASCIIZ name of the file or directory. The FSD does not need to validate this pointer.

iCurDirEnd is the index of the end of the current directory in pFile.

This is used to optimize FSD path processing. If iCurDirEnd == -1, there is no current directory relevant to the name text, that is, a device.

Remarks

The files specified are deleted.

The deletion of a file opened in DOS mode by the same process requesting the delete is supported. OS/2 calls FS_CLOSE for the file before calling FS_DELETE .

The file name may not contain wildcard characters.

FS_DOPAGEIO - Perform paging I/O operations

Purpose

Performs all the I/O operations in a PageCmdList.

Calling Sequence

```
int far pascal FS_DOPAGEIO(psffsi, psffsd, pList)

struct sffsi far * psffsi;
struct sffsd far * psffsd;
struct PageCmdHeader far * pList;
```

Where

psffsi is a pointer to the file-system-independent portion of an open file instance.

psffsd is a pointer to the file-system-dependent portion of an open file instance.

pList is a pointer to a PageCmdHeader structure.

The PageCmdHeader structure has the following format:

```
struct PageCmdHeader {
    unsigned char  InFlags;      /* Input Flags                         */
    unsigned char  OutFlags;     /* Output Flags - must be 0 on entry   */
    unsigned char  OpCount;      /* Number of operations                */
    unsigned char  Pad;          /* Pad for DWORD alignment             */
    unsigned long  Reserved1;    /* Currently Unused                    */
    unsigned long  Reserved2;    /* Currently Unused                    */
    unsigned long  Reserved3;    /* Currently Unused                    */
    struct PageCmd PageCmdList;  /* Currently Unused                    */
}
```

The PageCmd structure has the following format:

```
struct PageCmd {
    unsigned char Cmd;           /* Cmd Code (Read,Write,Verify)        */
    unsigned char Priority;      /* Same values as for req packets      */
    unsigned char Status;        /* Status byte                         */
    unsigned char Error;         /* I24 error code                      */
    unsigned long Addr;          /* Physical(0:32) or Virtual(16:16)    */
    unsigned long FileOffset;    /* Byte Offset in page file  */
}
```

Remarks

FS_DOPAGEIO performs all the I/O operations specified in the PageCmdList.

If the disk driver Extended Strategy requests, a request list will be built from the PageCmdList and issued to the driver.

If the disk driver does not support Extended Strategy requests, the FSD can either let the kernel do the emulation (See FS_OPENPAGEFILE to set this state) or has the option to do the emulation itself.

For a detailed description of the Extended Strategy request interface please see the *OS/2 Version 2.0 Physical Device Driver Reference* .

FS_EXIT - End of process

Purpose

Release FSD resources still held after process termination.

Calling Sequence

```
void far pascal FS_EXIT(uid, pid, pdb);

unsigned short uid;
unsigned short pid;
unsigned short pdb;
```

Where

uid is the user ID of the process. This will be a valid value.

pid is the process ID of the process. This will be a valid value.

pdb is the DOS mode process ID of the process. This will be a valid value .

Remarks

Because all files are closed when a process terminates, this call is not needed to release file resources. It is, however, useful if resources are being held due to unterminated searches (as in searches initiated from the DOS mode).

FS_FILEATTRIBUTE - Query/Set File Attribute

Purpose

Query/Set the attribute of the specified file.

Calling Sequence

```
int far pascal FS_FILEATTRIBUTE(flag, pcdfsi, pcdfsd, pName, iCurDirEnd,
                                pAttr)

unsigned short flag;
struct cdfsi far * pcdfsi;
struct cdfsd far * pcdfsd;
char far * pName;
unsigned short iCurDirEnd;
unsigned short far * pAttr;
```

Where

flag indicates retrieval or setting of attributes, with:

flag == 0 indicates retrieving the attribute.
flag == 1 indicates setting the attribute.
flag == all other values, reserved.

The value of flag passed to the FSD will be valid.

pcdfsi is a pointer to the file-system independent portion of an open file instance.

pcdfsd is a pointer to the file-system dependent portion of an open file instance.

pName is a pointer to the ASCIIZ name of the file or directory.

The FSD does not need to validate this pointer.

iCurDirEnd is the index of the end of the current directory in pName.

This is used to optimize FSD path processing. If iCurDirEnd == -1, there is no current directory relevant to the name text, that is, a device.

pAttr is a pointer to the attribute.

For flag == 0, the FSD should store the attribute in the indicated location .
For flag == 1, the FSD should retrieve the attribute from this location and set it in the file or directory.

The FSD does not need to validate this pointer.

Remarks

None

FS_FILEINFO - Query/Set a File's Information

Purpose

Returns information for a specific file.

Calling Sequence

```
int far pascal FS_FILEINFO(flag, psffsi, psffsd, level, pData, cbData,
                           IOflag)

unsigned short flag;
struct sffsi far * psffsi;
struct sffsd far * psffsd;
unsigned short level;
char far * pData;
unsigned short cbData;
unsigned short IOflag;
```

Where

flag indicates retrieval or setting of information.

flag == 0 indicates retrieving information.
flag == 1 indicates setting information.
All other values are reserved.

The value of flag passed to the FSD will be valid.

psffsi is a pointer to the file-system-independent portion of an open file instance.

psffsd is a pointer to the file-system-dependent portion of an open file instance.

level is the information level to be returned.

Level selects among a series of data structures to be returned.

pData is the address of the application data area.

Addressing of this data area is validated by the kernel (see FSH_PROBEBUF) .

When retrieval (flag == 0) is specified, the FSD will place the information into the buffer.

When outputting information to a file (flag == 1), the FSD will retrieve that data from the application buffer.

cbData is the length of the application data area.

For flag == 0, this is the length of the data the application wishes to retrieve. If there is not enough room for the entire level of data to be returned, the FSD will return a BUFFER OVERFLOW error.

For flag == 1, this is the length of data to be applied to the file.

IOflag indicates information about the operation on the handle.

IOflag == 0x0010 indicates write-through.
IOflag == 0x0020 indicates no-cache.

Remarks

If setting the time/date/DOS attributes on a file:

oCopy the new time/date/DOS attributes into the SFT
oSet ST_PCREAT, ST_PWRITE, and ST_PREAD
oClear ST_SCREAT, ST_SWRITE, and ST_SREAD

Note: ALSO NEW FOR 2.0, it is suggested that the FSD copy the DOS file attributes from the directory entry into the SFT. This allows the FSD and the OS2 kernel to handle FCB opens more efficiently.

If querying the date/time/DOS attributes on a file, simply copy the date/time /DOS attributes from the directory entry into the SFT.

Of the information passed in IOflag, the write-through bit is a mandatory bit in that any data written to the block device must be put out on the medium before the device driver returns. The no-cache bit, on the other hand, is an advisory bit that says whether the data being transferred is worth caching or not.

Supported information levels are described in the *OS/2 Version 2.0 Control Program Programming Reference* .

FS_FILEIO - Multi-function file I/O

Purpose

Perform multiple lock, unlock, seek, read, and write I/O.

Calling Sequence

```
int far pascal FS_FILEIO (psffsi, psffsd, pCmdList, cbCmdList, poError,
                          IOflag)

struct sffsi far * psffsi;
struct sffsd far * psffsd;
char far * pCmdList;
unsigned short cbCmdList;
unsigned short far * poError;
unsigned short IOflag;
```

Where

psffsi is a pointer to the file-system-independent portion of an open file instance.

psffsd is a pointer to the file-system-dependent portion of an open file instance.

pCmdList is a pointer to a command list that contains entries indicating what commands will be performed.

Each individual operation (CmdLock, CmdUnlock, CmdSeek, CmdIO) is performed as atomic operations until all are complete or until one fails. CmdLock executes a multiple range lock as an atomic operation. CmdUnlock executes a multiple range unlock as an atomic operation. Unlike CmdLock, CmdUnlock cannot fail as long as the parameters to it are correct, and the calling application had done a Lock earlier, so it can be viewed as atomic.

The validity of the user address is not verified (see FSH_PROBEBUF).

For CmdLock, the command format is:

```
struct CmdLock {
    unsigned short Cmd = 0;    /* 0 for lock operations         */
    unsigned short LockCnt;    /* number of locks that follow   */
    unsigned long  TimeOut;    /* ms time-out for lock success  */
}
```

which is followed by a series of records of the following format:

```
struct Lock {
    unsigned short Share = 0;    /* 0 for exclusive, 1 for read-only  */
    long           Start;        /* start of lock region              */
    long           Length;       /* length of lock region             */
}
```

If a lock within a CmdLock causes a time-out, none of the other locks within the scope of CmdLock are in force, because the lock operation is viewed as atomic .

CmdLock.TimeOut is the count in milliseconds, until the requesting process is to resume execution if the requested locks are not available. If CmdLock .TimeOut == 0, there will be no wait. If CmdLock.TimeOut < 0xFFFFFFFF it is the number of milliseconds to wait until the requested locks become available . If CmdLock.TimeOut == 0xFFFFFFFF then the thread will wait indefinitely until the requested locks become available.

Lock.Share defines the type of access other processes may have to the file-range being locked. If its value == 0, other processes have No-Access to the locked range. If its value == 1, other process have Read-Only access to the locked range.

For CmdUnlock, the command format is:

```
struct CmdUnlock {
    unsigned short Cmd = 1;      /* 1 for unlock operations        */
    unsigned short UnlockCnt;    /* Number of unlocks that follow  */
}
```

which is followed by a series of records of the following format:

```
struct UnLock {
    long Start;                  /* start of locked region         */
    long Length;                 /* length of locked region        */
}
```

For CmdSeek, the command format is:

```
struct CmdSeek {
    unsigned short Cmd = 2;    /* 2 for seek operation         */
    unsigned short Method;     /* 0 for absolute               */
                               /* 1 for relative to current    */
                               /* 2 for relative to EOF        */
    long           Position;   /* file seek position or delta  */
    long           Actual;     /* actual position seeked to    */
}
```

For CmdIO, the command format is:

```
struct CmdIO {
    unsigned short Cmd;         /* 3 for read, 4 for write      */
    void far * Buffer;          /* pointer to the data buffer   */
    unsigned short BufferLen;   /* number of bytes requested    */
    unsigned short Actual;      /* number of bytes transferred  */
}
```

cbCmdList is the length in bytes of the command list.

poError is the offset within the command list of the command that caused the error.

This field has a value only when an error occurs.

The validity of the user address has not been verified (see FSH_PROBEBUF) .

IOflag indicates information about the operation on the handle.

IOflag == 0x0010 indicates write-through.
IOflag == 0x0020 indicates no-cache.

Remarks

This function provides a simple mechanism for combining the file I/O operations into a single request and providing improved performance, particularly in a networking environment.

File systems that do not have the FileIO bit in their attribute field do not see this call: The command list is parsed by the IFS router. The FSD sees only FS_CHGFILEPTR, FS_READ, FS_WRITE calls.

File systems that have the FileIO bit in their attribute field see this call in its entirety. The atomicity guarantee applies only to the commands themselves and not to the list as a whole.

Of the information passed in IOflag, the write-through bit is a mandatory bit in that any data written to the block device must be put out on the medium before the device driver returns. The no-cache bit, on the other hand, is an advisory bit that says whether the data being transferred is worth caching or not.

FS_FILELOCKS - Request a file record lock/unlock

Purpose

Locks and/or unlocks a range (record) in a opened file.

Calling Sequence

```
int far pascal FS_FILELOCKS(psffsi, psffsd, pUnLockRange, pLockRange, timeout,
                            flags)

struct sffsi far * psffsi;
struct sffsd far * psffsd;
struct filelock far * pUnLockRange;
struct filelock far * pLockRange;
unsigned long timeout;
unsigned long flags;
```

Where

psffsi is a pointer to the file-system-independent portion of an open file instance.

psffsd is a pointer to the file-system-dependent portion of an open file instance.

pUnLockRange is a pointer to a filelock structure, identifying the range of the file to be unlocked. The filelock structure has the following format:

```
struct filelock {
    unsigned long FileOffset;   /* offset where the lock/unlock begins  */
    unsigned long RangeLength;  /* length of region locked/unlocked     */
}
```

If RangeLength is zero, no unlocking is required.

pLockRange is a pointer to a filelock structure, identifying the range of the file to be locked. If RangeLength is zero, no locking is required.

timeout is the maximum time in milliseconds that the requester wants to wait for the requested ranges, if they are not immediately available.

flags is the bit mask which specifies what actions are to taken:

SHARE Bit 0 on indicates other processes can share access to this locked range. Ranges with SHARE bit on can overlap.

SHARE Bit 0 off indicates the current process has exclusive access to the locked range. A range with the SHARE bit off CANNOT overlap with any other lock range.

ATOMIC Bit 1 on indicates an atomic lock request. If the lock range equals the unlock range, an atomic lock will occur. If the ranges are not equal, an error will be returned.

All other bits (2-31) are reserved and must be zero.

Remarks

This entry point was added to support the 32-bit DosSetFileLocks API.

If the lock and unlock range lengths are both zero, an error, ERROR_LOCK_ VIOLATION will be returned to the caller. If only a lock is desired, pUnLockRange can be NULL or both FileOffset and RangeLength should be set to zero when the call is made. The opposite is true for an unlock.

When the atomic bit is not set, the unlock occurs first then the lock is performed. If an error occurs on the unlock, an error is returned and the lock is not performed. If an error occurs on the lock, an error is returned and the unlock remains in effect if one was requested. If the atomic bit is set and the unlock range equals the lock range and the unlock range has shared access but wants to change the access to exclusive access, the function is atomic. FSDs may not support atomic lock functions. If error ERROR_ATOMIC_LOCK_NOT_SUPPORTED is returned , the application should do an unlock and lock the range using nonatomic operations. The application should also be sure to refresh its internal buffers prior to making any modifications.

Closing a file with locks still in force causes the locks to be released in no defined order.

Terminating a process with a file open and having issued locks on that file causes the file to be closed and the locks to be released in no defined order.

The figure below describes the level of access granted when the accessed region is locked. The locked regions can be anywhere in the logical file. Locking beyond end-of-file is not an error. It is expected that the time in which regions are locked will be short. Duplicating the handle duplicates access to the locked regions. Access to the locked regions is not duplicated across the DosExecPgm system call. The proper method for using locks is not to rely on being denied read or write access, but attempting to lock the region desired and examining the error code.

Locked Access Table

Action	Exclusive Lock	Shared Lock
Owner read	Success	Success
Non-owner read	Return code, not block	Success
Owner write	Success	Return code, not block
Non-owner write	Return code, not block	Return code, not block

The locked access table has the actions on the left as to whether owners or non-owners of a file do either reads or writes of files that have exclusive or shared locks set. A range to be locked for exclusive access must first be cleared of any locked subranges or locked any locked subranges or locked overlapping ranges.

FS_FINDCLOSE - Directory Read (Search) Close

Purpose

Provides the mechanism for an FSD to release resources allocated on behalf of FS_FINDFIRST and FS_FINDNEXT.

Calling Sequence

```
int far pascal FS_FINDCLOSE(pfsfsi, pfsfsd)

struct fsfsi far * pfsfsi;
struct fsfsd far * pfsfsd;
```

Where

pfsfsi is a pointer to the file-system-independent file search structure.

The FSD should not update this structure.

pfsfsd is a pointer to the file-system-dependent file search structure.

The FSD may use this to store information about continuation of its search .

Remarks

DosFindClose has been called on the handle associated with the search buffer . Any file system related information may be released.

If FS_FINDFIRST for a particular search returns an error, an FS_FINDCLOSE for that search will not be issued.

FS_FINDFIRST - Find First Matching File Name

Purpose

Find first occurrence of a file name in a directory.

Calling Sequence

```
int far pascal FS_FINDFIRST(pcdfsi, pcdfsd, pName, iCurDirEnd, attr, pfsfsi,
                            pfsfsd, pData, cbData, pcMatch, level, flags)

struct cdfsi far * pcdfsi;
struct cdfsd far * pcdfsd;
char far * pName;
unsigned short iCurDirEnd;
unsigned short attr;
struct fsfsi far * pfsfsi;
struct fsfsd far * pfsfsd;
char far * pData;
unsigned short cbData;
unsigned short far * pcMatch;
unsigned short level;
unsigned short flags;
```

Where

pcdfsi is a pointer to the file-system-independent working directory structure.

pcdfsd is a pointer to the file-system-dependent working directory structure .

pName is a pointer to the ASCIIZ name of the file or directory.

Wildcard characters are allowed only in the last component. The FSD does not need to validate this pointer.

iCurDirEnd is the index of the end of the current directory in pName.

This is used to optimize FSD path processing. If iCurDirEnd == -1 there is no current directory relevant to the name text, that is, a device.

attr is a bit field that governs the match.

Any directory entry whose attribute bit mask is a subset of attr and whose name matches that in pName should be returned. For example, an attribute of system and hidden is passed in. A file with the same name and an attribute of system is found. This file is returned. A file with the same name and no attributes (a regular file) is also returned. The attributes read-only and file- archive will not be passed in and should be ignored when comparing directory attributes .

The value of attr passed to the FSD will be valid. The bit 0x0040 indicates a non-8.3 filename format. It should be treated the same way as system and hidden attributes are.

pfsfsi is a pointer to the file-system-independent file-search structure.

The FSD should not update this structure.

pfsfsd is a pointer to the file-system-dependent file-search structure.

The FSD may use this to store information about continuation of the search .

pData is the address of the application data area.

Addressing of this data area is not validated by the kernel (see FSH_PROBEBUF). The FSD will fill in this area with a set of packed, variable-length structures that contain the requested data and matching file name.

cbData is the length of the application data area in bytes.

pcMatch is a pointer to the number of matching entries.

The FSD returns, at most, this number of entries; the FSD returns in this parameter the number of entries actually placed in the data area.

The FSD does not need to validate this pointer.

level is the information level to be returned.

Level selects among a series of data structures to be returned. The level passed to the FSD is valid.

flags indicates whether to return file-position information.

flags == 0 indicates that file-position information should not be returned and the information format described under DosFindFirst should be used.
flags == 1 indicates that file-position information should be returned and the information format described below should be used.

The flag passed to the FSD has a valid value.

Remarks

For flags == 1, the FSD must store in the first DWORD of the per-file attributes structure adequate information to allow the search to be resumed from the file by calling FS_FINDFROMNAME. For example, an ordinal representing the file 's position in the directory could be stored. If the filename must be used to restart the search, the DWORD may be left blank.

For level 0x0001 and flags == 0, directory information for FS_FINDFIRST is returned in the following format:

```
struct FileFindBuf {
    unsigned short dateCreate;
    unsigned short timeCreate;
    unsigned short dateAccess;
    unsigned short timeAccess;
    unsigned short dateWrite;
    unsigned short timeWrite;
    long           cbEOF;
    long           cbAlloc;
    unsigned short attr;
    unsigned char  cbName;
    unsigned char  szName[];
}
```

For level 0x0001 and flags == 1, directory information for FS_FINDFIRST is returned in the following format:

```
struct FileFromFindBuf {
    long           position;     /* position given to FSD on following */
                                 /* FS_FINDFROMNAME call               */
    unsigned short dateCreate;
    unsigned short timeCreate;
    unsigned short dateAccess;
    unsigned short timeAccess;
    unsigned short dateWrite;
    unsigned short timeWrite;
    long           cbEOF;
    long           cbAlloc;
    unsigned short attr;
    unsigned char  cbName;
    unsigned char  szName[];
}
```

The other information levels have similar format, with the position the first field in the structure for flags == 1.

If the non-8.3 filename format bit is set in the attributes of a file found by FS_FINDFIRST/NEXT/FROMNAME, it must be turned off in the copy of the attributes returned in pData.

If FS_FINDFIRST for a particular search returns an error, an FS_FINDCLOSE for that search will not be issued.

Sufficient information to find the next matching directory entry must be saved in the fsfsd data structure.

In the case where directory entry information overflows the pData area, the FSD should be able to continue the search from the entry which caused the overflow on the next FS_FINDNEXT or FS_FINDFROMNAME.

In the case of a global search in a directory, the first two entries in that directory as reported by the FSD should be '.' and '..' (current and the parent directories.

The example above just shows the effect of flags == 1 on a level 1 filefind record; level 2 and level 3 filefind records are similarly affected.

Note: The FSD will be called with the FINDFIRST/FINDFROMNAME interface when the 32-bit DosFindFirst/DosFindNext APIs are called. THIS IS A CHANGE FROM 1.X IFS interface for redirector FSDs. The kernel will also be massaging the find records so that they appear the way the caller expects. Redirectors who have to resume searches should take this information into account. (i.e . You might want to reduce the size of the buffer sent to the server, so that the position fields can be added to the beginning of all the find records).

FS_FINDFROMNAME - Find matching file name starting from name

Purpose

Find occurrence of a file name in a directory starting from a position or name.

Calling Sequence

```
int far pascal FS_FINDFROMNAME(pfsfsi, pfsfsd, pData, cbData, pcMatch, level,
                               position, pName, flags)

struct fsfsi far * pfsfsi;
struct fsfsd far * pfsfsd;
char far * pData;
unsigned short cbData;
unsigned short far * pcMatch;
unsigned short level;
unsigned long position;
char far * pName;
unsigned short flags;
```

Where

pfsfsi is a pointer to the file-system-independent file search structure. The FSD should not update this structure.

pfsfsd is a pointer to the file-system-dependent file search structure. The FSD may use this to store information about continuation of the search.

pData is the address of the application data area.

Addressing of this data area has not been validated by the kernel (see FSH_ PROBEBUF). The FSD will fill in this area with a set of packed, variable- length structures that contain the requested data and matching file names in the format required for DosFindFirst/DosFindNext.

cbData is the length of the application data area in bytes.

pcMatch is a pointer to the number of matching entries. The FSD will return at most this number of entries. The FSD will store into it the number of entries actually placed in the data area. The FSD does not need to validate this pointer.

level is the information level to be returned. Level selects among a series of structures of data to be returned. The level passed to the FSD is valid .

position is the file-system-specific information about where to restart the search from. This information was returned by the FSD in the ResultBuf for an FS_ FINDFIRST/FS_FINDNEXT/FS_FINDFROMNAME call.

pName is the filename from which to continue the search. The FSD does not need to validate this pointer.

flags indicates whether to return file position information. The flag passed to the FSD has a valid value.

Remarks

The FSD may use the position or filename to determine the position > from which to resume the directory search. The FSD need not return position if it uses name and vice versa.

For flags == 1, the FSD must store in the position field adequate information to allow the search to be resumed from the file by calling FS_FINDFROMNAME. See FS_FINDFIRST for a description of the data format.

The FSD must ensure that enough information is stored in the fsfsd data structure to enable it to continue the search.

Note: The FSD will be called with the FINDFIRST/FINDFROMNAME interface when the 32-bit DosFindFirst/DosFindNext APIs are called. THIS IS A CHANGE FROM 1.X IFS interface for redirector FSDs. The kernel will also be massaging the find records so that they appear the way the caller expects. Redirectors who have to resume searches should take this information into account. (i.e . You might want to reduce the size of the buffer sent to the server, so that the position fields can be added to the beginning of all the find records).

FS_FINDNEXT - Find next matching file name.

Purpose

Find the next occurrence of a file name in a directory.

Calling Sequence

```
int far pascal FS_FINDNEXT(pfsfsi, pfsfsd, pData, cbData, pcMatch, level,
                           flags)

struct fsfsi far * pfsfsi;
struct fsfsd far * pfsfsd;
char far * pData;
unsigned short cbData;
unsigned short far * pcMatch;
unsigned short level;
unsigned short flags;
```

Where

pfsfsi is a pointer to the file-system-independent file-search structure. The FSD should not update this structure.

pfsfsd is a pointer to the file-system-dependent file-search structure. The FSD may use this to store information about continuation of the search.

pData is the address of the application area.

Addressing of this data area is not validated by the kernel (see FSH_PROBEBUF). The FSD fills in this area with a set of packed, variable-length structures that contain the requested data and matching file names.

cbData is the length of the application data area in bytes.

pcMatch is a pointer to the number of matching entries.

The FSD returns, at most, this number of entries. The FSD returns the the number of entries actually placed in the data area in this parameter.

The FSD does not need to validate this pointer.

level is the information level to be returned. Level selects among a series of structures of data to be returned. The level passed to the FSD is valid .

flags indicates whether to return file-position information.

Remarks

For flags == -1, the FSD must store in the position field adequate information to allow the search to be resumed from the file by calling FS_FINDFROMNAME. See FS_FINDFIRST for a description of the data format.

The level passed to FS_FINDNEXT is the same level as that passed to FS_ FINDFIRST to initiate the search.

Sufficient information to find the next matching directory entry must be saved in the fsfsd data structure.

The FSD should take care of the case where the pData area overflow may occur . FSDs should be able to start the search from the same entry for the next FS_ FINDNEXT as the one for which the overflow occurred.

In the case of a global search in a directory, the first two entries in that directory as reported by the FSD should be '.' and '..' (current and parent directories).

FS_FINDNOTIFYCLOSE - Close Find-Notify Handle

Purpose

Closes the association between a Find-Notify handle and a DosFindNotifyFirst or DosFindNotifyNext function.

Calling Sequence

```
int far pascal FS_FINDNOTIFYCLOSE(handle)

unsigned short handle;
```

Where

handle is the directory handle.

This handle was returned by the FSD on a previous FS_FINDNOTIFYFIRST or FS_ FINDNOTIFYNEXT call.

Remarks

Provides the mechanism for an FSD to release resources allocated on behalf of FS_FINDNOTIFYFIRST and FS_FINDNOTIFYNEXT.

FS_FINDNOTIFYFIRST returns a handle to the find-notify request. FS_ FINDNOTIFYCLOSE closes the handle associated with that find-notify request and releases file system information related to that handle.

FS_FINDNOTIFYFIRST - Monitor a directory for changes.

Purpose

Start monitoring a directory for changes.

Calling Sequence

```
int far pascal FS_FINDNOTIFYFIRST(pcdfsi, pcdfsd, pName, iCurDirEnd, attr,
                                  pHandle, pData, cbData, pcMatch, level,
                                  timeout)

struct cdfsi far * pcdfsi;
struct cdfsd far * pcdfsd;
char far * pName;
unsigned short iCurDirEnd;
unsigned short attr;
unsigned short far * pHandle; char far * pData;
unsigned short cbData;
unsigned short far * pMatch;
unsigned short level;
unsigned long timeout;
```

Where

pcdfsi is a pointer to the file-system-independent working directory structure.

pcdfsd is a pointer to the file-system-dependent working directory structure .

pName is a pointer to the ASCIIZ name of the file or directory.

Wildcard characters are allowed only in the last component. The FSD does not need to verify this pointer.

iCurDirEnd is the index of the end of the current directory in pName.

This is used to optimize FSD path processing. If iCurDirEnd == -1 there is no current directory relevant to the name text, that is, a device.

attr is the bit field that governs the match.

Any directory entry whose attribute bit mask is a subset of attr and whose name matches that in pName should be returned. See FS_FINDFIRST for an explanation.

pHandle is a pointer to the handle for the find-notify request.

The FSD allocates a handle for the find-notify request, that is, a handle to the directory monitoring continuation information, and stores it here. This handle is passed to FS_FINDNOTIFYNEXT to continue directory monitoring.

The FSD does not need to verify this pointer.

pData is the address of the application data area.

Addressing of this data area is not validated by the kernel (see FSH_PROBEBUF). The FSD fills in this area with a set of packed, variable-length structures that contain the requested data and matching file names.

cbData is the length of the application data area in bytes.

pcMatch is a pointer to the number of matching entries.

The FSD returns, at most, this number of entries. The FSD returns in this parameter the number of entries actually placed in the data area.

The FSD does not need to verify this pointer.

level is the information level to be returned.

Level selects among a series of data structures to be returned. See the description of DosFindNotifyFirst in the *OS/2 Version 2.0 Control Program Programming Reference* for more information.

The level passed to the FSD is valid.

timeout is the time-out in milliseconds.

The FSD waits until either the time-out has expired, the buffer is full, or the specified number of entries has been returned before returning to the caller .

Remarks

None.

FS_FINDNOTIFYNEXT - Resume reporting directory changes

Purpose

Resume reporting of changes to a file or directory.

Calling Sequence

```
int far pascal FS_FINDNOTIFYNEXT(handle, pData, cbData, pcMatch, level,
                                 timeout)

unsigned short handle;
char far * pData;
unsigned short cbData;
unsigned short far * pcMatch;
unsigned short level;
unsigned long timeout;
```

Where

handle is the handle to the find-notify request.

This handle was returned by the FSD and is associated with a previous FS_ FINDNOTIFYFIRST or FS_FINDNOTIFYNEXT call.

pData is the address of the application data area.

Addressing of this data area is not validated by the kernel (see FSH_PROBEBUF). The FSD fills in this area with a set of packed, variable-length structures that contain the requested data and matching file names.

cbData is the length of the application data area in bytes.

pcMatch is a pointer to the number of matching entries.

The FSD returns, at most, this number of entries. The FSD returns in this parameter the number of entries actually placed in the data area.

The FSD does not need to verify this pointer.

level is the information level to be returned.

Level selects among a series of data structures to be returned. See the description of DosFindNotifyFirst in the *OS/2 Version 2.0 Control Program Programming Reference* for more information.

The level passed to the FSD is valid.

timeout is the time-out in milliseconds.

The FSD waits until either the time-out has expired, the buffer is full, or the specified number of entries has been returned before returning to the caller .

Remarks

pcMatch is the number of changes required to directories or files that match the pName target and attr specified during a related, previous FS_FINDNOTIFYFIRST . The file system uses this field to return the number of changes that actually occurred since the issue of the present FS_FINDNOTIFYNEXT.

The level passed to FS_FINDNOTIFYNEXT is the same level as that passed to FS_ FINDNOTIFYFIRST to initiate the search.

FS_FLUSHBUF - Commit file buffers

Purpose

Flushes cache buffers for a specific volume.

Calling Sequence

```
int far pascal FS_FLUSHBUF(hVPB, flag)

unsigned short hVPB;
unsigned short flag;
```

Where

hVPB is the handle to the volume for flush.

flag is used to indicate discarding of cached data.

flag == 0 indicates cached data may be retained.
flag == 1 indicates the FSD will discard any cached data after flushing it to the specified volume.

All other values are reserved.

Remarks

None.

FS_FSCTL - File System Control

Purpose

Allow an extended standard interface between an application and a file system driver.

Calling Sequence

```
int far pascal FS_FSCTL(pArgdat, iArgType, func, pParm, lenParm, plenParmIO,
                        pData, lenData, plenDataIO)

union argdat far * pArgDat;
unsigned short iArgType;
unsigned short func;
char far * pParm;
unsigned short lenParm;
unsigned short far * plenParmIO;
char far * pData;
unsigned short lenData;
unsigned short far * plenDataIO;
```

Where

pArgDat is a pointer to the union whose contents depend on iArgType. The union is defined as follows:

```
union argdat {

    /* pArgType = 1, FileHandle directed case */

    struct sf {
        struct sffsi far * psffsi;
        struct sffsd far * psffsd;
    };

    /* pArgType = 2, Pathname directed case */

    struct cd {
        struct cdfsi far * pcdfsi;
        struct cdfsd far * pcdfsd;
        char far *         pPath;
        unsigned short     iCurDirEnd;
    };

    /* pArgType = 3, FSD Name directed case */
    /* pArgDat is Null                      */
};
```

iArgType indicates the argument type.

iArgType = 1
means that pArgDat->sf.psffsi and pArgDat->sf.psffsd point to an sffsi and sffsd, respectively.

iArgType = 2
means that pArgDat->cd.pcdfsi and pArgDat->cd.pcdfsd point to a cdfsi and cdfsd, pArgDat->cd.pPath points to a canonical pathname, and pArgDat->cd .iCurDirEnd gives the index of the end of the current directory in pPath. The FSD does not need to verify the pPath pointer.

iArgType = 3
means that the call was FSD name routed, and pArgDat is a NULL pointer.

func indicates the function to perform.

func == 1 indicates a request for new error code information.
func == 2 indicates a request for the maximum EA size and EA list size supported by the FSD.

pParm is the address of the application input parameter area.

Addressing of this data area has not been validated by the kernel (see FSH_ PROBEBUF).

lenParm is the maximum length of the application parameter area (pParm).

plenParmIO On input, contains the length in bytes of the parameters being passed in to the FSD in pParm. On return, contains the length in bytes of data returned in pParm by the FSD. The length of the data returned by the FSD in pParm must not exceed the length in lenParm. Addressing of this area is not validated by the kernel (see FSH_PROBEBUF).

pData is the address of the application output data area.

Addressing of this data area is not validated by the kernel (see FSH_PROBEBUF).

lenData is the maximum length of the application output data area (pData) .

plenDataIO On input, contains the length in bytes of the data being passed in to the FSD in pData. On return, contains the length in bytes of data returned in pData by the FSD. The length of the data returned by the FSD in pData must not exceed the length in lenData. Addressing of this area is not validated by the kernel (see FSH_PROBEBUF).

Remarks

The accessibility of the parameter and data buffers has not been validated by the kernel. FS_PROBEBUF must be used.

All FSDs must support func == 1 to return new error code information and func == 2 to return the limits of the EA sizes.

For func == 1, the error code is passed to the FSD in the first WORD of the parameter area. On return, the first word of the data area contains the length of the ASCIIZ string containing an explanation of the error code. The data area contains the ASCIIZ string beginning at the second WORD.

For func == 2, the maximum EA and EA list sizes supported by the FSD are returned in the buffer pointed to by pData in the following format:

```
struct EASizeBufStruc {
    unsigned short easb_MaxEASize;      /* Max size of an individual EA */
    unsigned long easb_MaxEAListSize;   /* Max full EA list size        */
}
```

FS_FSINFO - File System Information

Purpose

Returns or sets information for a file system device.

Calling Sequence

```
int far pascal FS_FSINFO(flag, hVPB, pData, cbData, level)

unsigned short flag;
unsigned short hVPB;
char far * pData;
unsigned short cbData;
unsigned short level;
```

Where

flag indicates retrieval or setting of information.

flag == 0 indicates retrieving information.
flag == 1 indicates setting information on the media.
All other values are reserved.

hVPB is the handle to the volume of interest.

pData is the address of the application output data area.

Addressing of this data area has not been validated by the kernel (see (FSH_ PROBEBUF).

cbData is the length of the application data area.

For flag == 0, this is the length of the data the application wishes to retrieve. If there is not enough room for the entire level of data to be returned, the FSD will return a BUFFER OVERFLOW error. For flag == 1, this is the length of the data to be sent to the file system.

level is the information level to be returned.

Level selects among a series of structures of data to be returned or set. See DosQFSInfo and DosSetFSInfo for information.

Remarks

None.

FS_INIT - File system driver initialization

Purpose

Request file system driver initialization.

Calling Sequence

```
int far pascal FS_INIT(szParm, DevHelp, pMiniFSD)

char far * szParm;
unsigned long DevHelp;
unsigned long far * pMiniFSD;
```

Where

szParm is a pointer to the ASCIIZ parameters following the CONFIG.SYS IFS = command that loaded the FSD. The FSD does not need to verify this pointer .

DevHelp is the address of the kernel entry point for the DevHelp routines .

This is used exactly as the device driver DevHelp address, and can be used by an FSD that needs access to some of the device helper services.

pMiniFSD is a pointer to data passed between the mini-FSD and the FSD, or null.

Remarks

This call is made during system initialization to allow the FSD to perform actions necessary for beginning operation. The FSD may successfully initialize by returning 0 or may reject installation (invalid parameters, incompatible hardware, etc .) by returning the appropriate error code. If rejection is selected, all FSD selectors and segments are released.

pMiniFSD will be null, except when booting from a volume managed by an FSD and the exported name of the FSD matches the exported name of the mini-FSD. In this case, pMiniFSD will point to data established by the mini-FSD (See MFS_INIT) .

FS_IOCTL - I/O Control for Devices

Purpose

Perform control function on the device specified by the opened device handle .

Calling Sequence

```
int far pascal FS_IOCTL(psffsi, psffsd, cat, func, pParm, lenMaxParm,
                        plenInOutParm, pData, lenMaxData, plenInOutData)

struct sffsi far * psffsi;
struct sffsd far * psffsd;
unsigned short cat;
unsigned short func;
char far * pParm;
unsigned short lenMaxParm;
unsigned short * plenInOutParm;
char far * pData;
unsigned short lenMaxData;
unsigned short * plenInOutData;
```

Where

psffsi is a pointer to the file-system-independent portion of an open file instance.

psffsd is a pointer to the file-system-dependent portion of an open file instance.

cat is the category of the function to be performed.

func is the function within the category to be performed.

pParm is the address of the application input parameter area.

Addressing of this data area is not validated by the kernel (see FSH_PROBEBUF). A null value indicates that the parameter is unspecified for this function . lenMaxParm is the byte length of the application input parameter area.

If lenMaxParm is 0, *plenInOutParm is 0, and pParm is not null, it means that the data buffer length is unknown due to the request being submitted via an old IOCTL or DosDevIOCtl interface.

plenInOutParm is the pointer to an unsigned short that contains the length of the parameter area in use on input and is set by the FSD to be the length of the parameter area in use on output.

Addressing of this data area is not validated by the kernel (see FSH_PROBEBUF). A null value indicates that the parameter is unspecified for this function .

pData is the address of the application output data area.

Addressing of this data area has not been validated by the kernel (see FSH_ PROBEBUF). A null value indicates that the parameter is unspecified for this function.

lenMaxData is the byte length of the application output data area.

If lenMaxData is 0, *plenInOutData is 0, and pData is not null, it means that the data buffer length is unknown due to the request being submitted via an old IOCTL or DosDevIOCtl interface.

plenInOutData is the pointer to an unsigned short that contains the length of the data area in use on input and is set by the FSD to be the length of the data area in use on output.

Addressing of this data area is not validated by the kernel (see FSH_PROBEBUF). A null value indicates that the parameter is unspecified for this function .

Remarks

Note: This entry point's parameter list definition has changed from the 1.x IFS document. If the parameters plenInOutParm and plenInOutData are null, use the lenMax parameters as the buffer sizes sent to any file system helper.

FS_MKDIR - Make Subdirectory

Purpose

Create the specified directory.

Calling Sequence

```
int far pascal FS_MKDIR(pcdfsi, pcdfsd, pName, iCurDirEnd, pEABuf, flags)

struct cdfsi far * pcdfsi;
struct cdfsd far * pcdfsd;
char far * pName;
unsigned short iCurDirEnd;
char far * pEABuf;
unsigned short flags;
```

Where

pcdfsi is a pointer to the file-system-independent working directory structure.

pcdfsd is a pointer to the file-system-dependent working directory structure .

pName is a pointer to the ASCIIZ name of the directory to be created.

The FSD does not need to verify this pointer.

iCurDirEnd is the index of the end of the current directory in pName.

This is used to optimize FSD path processing. If iCurDirEnd == -1, there is no current directory relevant to the name text, that is, a device.

pEABuf is a pointer to the extended attribute buffer.

This buffer contains attributes that will be set upon creation of the new directory. If NULL, no extended attributes are to be set. Addressing of this data area has not been validated by the kernel (see FSH_PROBEBUF).

flags indicates the name type.

Flags == 0x0040 indicates a non-8.3 filename format. All other values are reserved.

Remarks

The FSD needs to do the time stamping itself. There is no aid in the kernel for time stamping sub-directories. FAT only supports creation time stamp and sets the other two fields to zeroes. An FSD should do the same. The FSD can obtain the current time/date from the infoseg.

A new directory called pName should be created if possible. The standard directory entries '.' and '..' should be put into the directory.

The non-8.3 filename format attribute in the directory entry should be set according to the value in flags.

FS_MOUNT - Mount/unmount volumes

Purpose

Examination of a volume by an FSD to see if it recognizes the file system format.

Calling Sequence

```
int far pascal FS_MOUNT(flag, pvpfsi, pvpfsd, hVPB, pBoot)

unsigned short flag;
struct vpfsi far * pvpfsi;
struct vpfsd far * pvpfsd;
unsigned short hVPB;
char far * pBoot;
```

Where

flag indicates operation requested.

flag == 0 indicates that the FSD is requested to mount or accept a volume .

flag == 1 indicates that the FSD is being advised that the specified volume has been removed.

flag == 2 indicates that the FSD is requested to release all internal storage assigned to that volume as it has been removed from its driver and the last kernel- managed reference to that volume has been removed.

flag == 3 indicates that the FSD is requested to accept the volume regardless of recognition in preparation for formatting for use with the FSD.

All other values are reserved.

The value passed to the FSD will be valid.

pvpfsi is a pointer to the file-system-independent portion of VPB.

If the media contains an OS/2-recognizable boot sector, then the vpi_vid field contains the 32-bit identifier for that volume. If the media does not contain such a boot sector, the FSD must generate a unique label for the media and place it into the vpi_vid field.

pvpfsd is a pointer to the file-system-dependent portion of VPB.

The FSD may store information as necessary into this area.

hVPB is the handle to the volume

pBoot is a pointer to sector 0 read from the media.

This pointer is only valid when flag == 0.The buffer the pointer refers to must not be modified. The pointer is always valid and does not need to be verified when flag == 0. If a read error occurred, the buffer will contain zeroes .

Remarks

The FSD examines the volume presented and determine whether it recognizes the file system. If it does, it returns zero, after having filled in appropriate parts of the vpfsi and vpfsd data structures. The vpi_vid and vpi_text fields must be filled in by the FSD. If the FSD has an OS/2 format boot sector, it must convert the label from the media into ASCIIZ form. The vpi_hDev field is filled in by OS/2. If the volume is unrecognized, the driver returns non-zero.

The vpi_text and vpi_vid must be updated by the FSD each time these values change.

The contents of the vpfsd data structure are as follows:

FLAG = 0 The FSD is expected to issue an FSD_FINDDUPHVPB to see if a duplicate VPB exists. If one does exist, the VPB fs dependent area of the new VPB is invalid and the new VPB will be unmounted after the FSD returns from the MOUNT. The FSD is expected to update the FS dependent area of the old duplicate VPB. If no duplicate VPB exists, the FSD should initialize the FS dependent area.

FLAG = 1 VPB FS dependent part is same as when FSD last modified it.

FLAG = 2 VPB FS dependent part is same as when FSD last modified it.

After media recognition time, the volume parameters may be examined using the FSH_GETVOLPARM call. The volume parameters should not be changed after media recognition time.

During a mount request, the FSD may examine other sectors on the media by using FSH_DOVOLIO to perform the I/O. If an uncertain-media return is detected, the FSD is expected to clean up and return an UNCERTAIN MEDIA error in order to allow the volume mount logic to restart on the newly-inserted media. The FSD must provide the buffer to use for additional I/O.

The OS/2 kernel manages the VPB through a reference count. All volume- specific objects are labeled with the appropriate volume handle and represent references to the VPB. When all kernel references to a volume disappear, FS_MOUNT is called with flag == 2, indicating a dismount request.

When the kernel detects that a volume has been removed from its driver, but there are still outstanding references to the volume, FS_MOUNT is called with flag == 1 to allow the FSD to drop clean (or other regenerable) data for the volume. Data which is dirty and cannot be regenerated should be kept so that it may be written to the volume when it is remounted in the drive.

When a volume is to be formatted for use with an FSD, the kernel calls the FSD's FS_MOUNT entry point with flag == 3 to allow the FSD to prepare for the format operation. The FSD should accept the volume even if it is not a volume of the type that FSD recognizes, since the point of format is to change the file system on the volume. The operation may fail if formatting does not make sense . (For example, an FSD which supports only CD-ROM.)

Since the hardware does not allow for kernel-mediated removal of media, it is certain that the unmount request is issued when the volume is not present in any drive .

FS_MOVE - Move a file or subdirectory

Purpose

Moves (renames) the specified file or subdirectory.

Calling Sequence

```
int far pascal FS_MOVE(pcdfsi, pcdfsd, pSrc, iSrcCurDirEnd, pDst,
                       iDstCurDirEnd, flags)

struct cdfsi far * pcdfsi;
struct cdfsd far * pcdfsd;
```

```
char far * pSrc;
unsigned short iSrcCurDirEnd;
char far * pDst;
unsigned short iDstCurDirEnd;
unsigned short flags;
```

Where

pcdfsi is a pointer to the file-system-independent working directory structure.

pcdfsd is a pointer to the file-system-dependent working directory structure .

pSrc is a pointer to the ASCIIZ name of the source file or directory.

The FSD does not need to verify this pointer.

iSrcCurDirEnd is the index of the end of the current directory in pSrc.

This is used to optimize FSD path processing. If iSrcCurDirEnd == -1 there is no current directory relevant to the source name text.

pDst is a pointer to the ASCIIZ name of the destination file or directory .

The FSD does not need to verify this pointer.

iDstCurDirEnd is the index of the end of the current directory in pDst.

This is used to optimize FSD path processing. If iDstCurDirEnd == -1 there is no current directory relevant to the destination name text.

flags indicates destination name type.

Flags == 0x0040 indicates non-8.3 filename format. All other values are reserved.

Remarks

The file specified in filename should be moved to or renamed as the destination filename, if possible.

Neither the source nor the destination filename may contain wildcard characters.

FS_MOVE may be used to change the case in filenames.

The non-8.3 filename format attribute in the directory entry for the destination name should be set according to the value in flags.

In the case of a subdirectory move, the system does the following checking :

oNo files in this directory or its subdirectories are open.
oThis directory or any of its subdirectories is not the current directory for any process in the system.

In addition, the system also checks for circularity in source and target directory names; that is, the source directory is not a prefix of the target directory .

Note: OS/2 does not validate input parameters. Therefore, an FSD should call FSH_PROBEBUF where appropriate.

FS_NEWSIZE - Change File's Logical Size

Purpose

Changes a file's logical (EOD) size.

Calling Sequence

```
int far pascal FS_NEWSIZE(psffsi, psffsd, len, IOflag)

struct sffsi far * psffsi;
struct sffsd far * psffsd;
unsigned long len;
unsigned short IOflag;
```

Where

psffsi is a pointer to the file-system-independent portion of an open file instance.

psffsd is a pointer to the file-system-dependent portion of an open file instance.

len is the desired new length of the file.

IOflag indicates information about the operation on the handle.

IOflag == 0x0010 indicates write-through.
IOflag == 0x0020 indicates no-cache.

Remarks

The FSD should return an error if an attempt is made to write beyond the end with a direct access device handle.

The file system driver attempts to set the size (EOD) of the file to newsize and update sfi_size, if successful. If the new size is larger than the currently allocated size, the file system driver arranges for for efficient access to the newly-allocated storage.

Of the information passed in IOflag, the write-through bit is a mandatory bit in that any data written to the block device must be put out on the medium before the device driver returns. The no-cache bit, on the other hand, is an advisory bit that says whether the data being transferred is worth caching or not.

FS_NMPIPE - Do a remote named pipe operation.

Purpose

Perform a special purpose named pipe operation remotely.

Calling Sequence

```
int far pascal FS_NMPIPE(psffsi, psffsd, OpType, pOpRec, pData, pName)

struct sffsi far * psffsi;
struct sffsd far * psffsd;
unsigned short OpType;
union npoper far * pOpRec;
char far * pData;
char far * pName;
```

Where

psffsi is a pointer to the file-system-independent portion of an open file instance.

psffsd is a pointer to the file-system-dependent portion of an open file instance.

OpType is the operation to be performed. This parameter has the following values:

```
NMP_GetPHandState     0x21
NMP_SetPHandState     0x01
NMP_PipeQInfo         0x22
NMP_PeekPipe          0x23
NMP_ConnectPipe       0x24
NMP_DisconnectPipe    0x25
NMP_TransactPipe      0x26
NMP_ReadRaw           0x11
NMP_WriteRaw          0x31
NMP_WaitPipe          0x53
NMP_CallPipe          0x54
NMP_QNmPipeSemState   0x58
```

pOpRec is the data record which varies depending on the value of OpType. The first parameter in each structure encodes the length of the parameter block . The second parameter, if non-zero, indicates that the pData parameter is supplied and gives its length. The following record formats are used:

```
union npoper {
    struct phs_param phs;
    struct npi_param npi;
    struct npr_param npr;
    struct npw_param npw;
    struct npq_param npq;
    struct npx_param npx;
    struct npp_param npp;
    struct npt_param npt;
    struct qnps_param qnps;
    struct npc_param npc;
    struct npd_param npd;
};

/* Get/SetPhandState parameter block */

struct phs_param {
    short phs_len;
    short phs_dlen;
    short phs_pmode; /* pipe mode set or returned */
};

/* DosQNmPipeInfo parameter block */
struct npi_param {
    short npi_len;
    short npi_dlen;
    short npi_level; /* information level desired */
};

/* DosRawReadNmPipe parameters */
/* data is buffer addr         */

struct npr_param {
    short npr_len;
    short npr_dlen;
    short npr_nbyt; /* number of bytes read */
};


/* DosRawWriteNmPipe parameters */
/* data is buffer addr          */

struct npw_param {
    short npw_len;
    short npw_dlen;
    short npw_nbyt; /* number of bytes written */
};

/* NPipeWait parameters */

struct npq_param {
    short npq_len;
    short npq_dlen;
    long npq_timeo;  /* time-out in milliseconds */
    short npq_prio;  /* priority of caller       */
};

/* DosCallNmPipe parameters */
/* data is in-buffer addr   */

struct npx_param {
    short npx_len;
    unsigned short npx_ilen;  /* length of in-buffer      */
    char far * npx_obuf;      /* pointer to out-buffer    */
    unsigned short npx_ilen;  /* length of out-buffer     */
    unsigned short npx_nbyt;  /* number of bytes read     */
    long npx_timeo;           /* time-out in milliseconds */
};

/* PeekPipe parameters, data is buffer addr */

struct npp_param {
    short npp_len;
    unsigned short npp_dlen;
    unsigned short npp_nbyt;   /* number of bytes read      */
    unsigned short npp_av10;   /* bytes left in pipe        */
    unsigned short npp_av11;   /* bytes left in current msg */
    unsigned short npp_state;  /* pipe state                */
};

/* DosTransactNmPipe parameters */
/* data is in-buffer addr       */

struct npt_param {
    short npt_len;
    unsigned short npt_ilen;  /* length of in-buffer   */
    char far * npt_obuf;      /* pointer to out-buffer */
    unsigned short npt_olen;  /* length of out-buffer  */
    unsigned short npt_nbyt;  /* number of bytes read  */
};

/* QNmPipeSemState parameter block */ /* data is user data buffer */

struct qnps_param {
    unsigned short qnps_len;  /* length of parameter block      */
    unsigned short qnps_dlen;  /* length of supplied data block */
    long qnps_semh;            /* system semaphore handle       */
    unsigned short qnps_nbyt;  /* number of bytes returned      */
};

/* ConnectPipe parameter block, no data block */

struct npc_param {
    unsigned short npc_len;   /* length of parameter block */
    unsigned short npc_dlen;  /* length of data block      */
};

/* DisconnectPipe parameter block, no data block */

struct npd_param {
    unsigned short npd_len;   /* length of parameter block */
    unsigned short npd_dlen;  /* length of data block      */
};
```

pData is a pointer to a user data for operations which require it. When the pointer is supplied, its length will be given by the second element of the pOpRec structure.

pName is a pointer to a remote pipe name. Supplied only for NMP_WAITPIPE and NMP_CALLPIPE operations. For these two operations only, the psffsi and psffsd parameters have no significance.

Remarks

This entry point is for support of special remote named pipe operations. Not all pointer parameters are used for all operations. In cases where a particular pointer has no significance, it will be NULL.

This entry point will be called only for the UNC FSD. Non-UNC FSDs are required to have this entry point, but should return NOT SUPPORTED if called.

FS_OPENCREATE - Open a file

Purpose

Opens (or creates) the specified file.

Calling Sequence

```
int far pascal FS_OPENCREATE(pcdfsi, pcdfsd, pName, iCurDirEnd, psffsi,
                             psffsd, ulOpenMode, usOpenFlag, pusAction,
                             usAttr, pcEABuf, pfgenflag)

struct cdfsi far * pcdfsi;
struct cdfsd far * pcdfsd;
char far * pName;
unsigned short iCurDirEnd;
struct sffsi far * psffsi;
struct sffsd far * psffsd;
unsigned long ulOpenMode;
unsigned short usOpenFlag;
unsigned short far * pusAction;
unsigned short usAttr;
char far * pcEABuf;
unsigned short far * pfgenflag;
```

Where

pcdfsi is a pointer to the file-system-independent working directory structure.

The contents of this structure are invalid for direct access opens.

pcdfsd is a pointer to the file-system-dependent working directory structure . The contents of this structure are invalid for direct access opens. For remote character devices, this field contains a pointer to a DWORD that was obtained from the remote FSD when the remote device was attached to this FSD. The FSD can use this DWORD to identify the remote device.

pName is a pointer to the ASCIIZ name of the file to be opened.

The FSD does not need to verify this pointer.

iCurDirEnd is the index of the end of the current directory in pName.

This is used to optimize FSD path processing. If iCurDirEnd == -1, there is no current directory relevant to the name text, that is a device. This value is invalid for direct access opens.

psffsi is a pointer to the file-system-independent portion of an open file instance.

psffsd is a pointer to the file-system-dependent portion of an open file instance.

ulOpenMode indicates the desired sharing mode and access mode for the file handle.

See *OS/2 Version 2.0 Control Program Programming Reference* for a description of the OpenMode parameter for DosOpen.

An additional access mode 3 is defined when the file is being opened on behalf of OS/2, loaded for purposes of executing a file or loading a module. If the file system does not support an executable attribute, it should treat this access mode as open for reading. The value of ulOpenMode passed to the FSD will be valid.

usOpenFlag indicates the action taken when the file is present or absent.

See *OS/2 Version 2.0 Control Program Programming Reference* for a description of the usOpenFlag parameter for DosOpen.

The value of openflag passed to the FSD is valid. This value is invalid for direct access opens.

pusAction is the location where the FSD returns a description of the action taken as governed by openflag.

The FSD does not need to verify this pointer. The contents of Action are invalid on return for direct access opens.

usAttr are the OS/2 file attributes.

This value is invalid for direct access opens.

pcEABuf is a pointer to the extended attribute buffer.

This buffer contains attributes that will be set upon creation of a new file or upon replacement of an existing file. If NULL, no extended attributes are to be set. Addressing of this data area has not been validated by the OS/2 kernel (see FSH_PROBEBUF). The contents of EABuf are invalid on return for direct access opens.

pfgenflag is a pointer to an unsigned short of flags returned by the FSD. The only flag currently defined is 0x0001 fGenNeedEA, which indicates that there are critical EAs associated with the file. The FSD does not need to verify this pointer.

Remarks

For the file create operation, if successful, ST_CREAT and ST_PCREAT are set . This causes the file to have zero as last read and last write time. If the last read/write time stamps are to be the same as the create time, simply set ST_ SWRITE, ST_PWRITE, ST_SREAD, and ST_PREAD as well.

For the file open operation, the FSD copies all supported time stamps from the directory entry into the SFT.

Note: ALSO NEW FOR 2.0, it is suggested that the FSD copy the DOS file attributes from the directory entry into the SFT. This allows the FSD and the OS2 kernel to handle FCB opens more efficiently.

The sharing mode may be zero if this is a request to open a file from the DOS mode or for an FCB request.

FCB requests for read-write access to a read-only file are mapped to read- only access and reflected in the sfi_mode field by the FSD. An

FCB request is indicated by the third bit set in the sfi_type field.

The flags defined for the sfi_type field are:

otype == 0x0000 indicates file.
otype == 0x0001 indicates device.
otype == 0x0002 indicates named pipe.
otype == 0x0004 indicates FCB open.
oAll other values are reserved.

FSDs are required to initialize the sfi_type field, preserving the FCB bit .

On entry, the sfi_hvpb field is filled in. If the file's logical size (EOD) is specified, it is passed in the sfi_size field. To the extent possible, the file system tries to allocate this much storage for efficient access.

Extended attributes are set for:

othe creation ofa new file
othe truncation of an existing file
othe replacement of an existing file.

They are not set for a normal open.

If the standard OS/2 file creation attributes are specified, they are passed in the attr field. To the extent possible, the file system interprets the extended attributes and applies them to the newly-created or existing file. Extended attributes (EAs) that the file system does not itself use are retained with the file and not discarded or rejected.

When replacing an existing file, the FSD should not change the case of the existing file.

FSDs are required to support direct access opens. These are indicated by a bit set in the sffsi.sfi_mode field. See *OS/2 Version 2.0 Control Program Programming Referenc*e for more information on DosOpen. Some of the parameters passed to the FSD for direct access opens are invalid, as described above.

On a successful return, the following fields in the sffsi structure must be filled in by the file system driver: sfi_size and all the time and date fields .

The file-system-dependent portion of an open file instance passed to the FSD for FS_OPENCREATE is never initialized.

Infinite FCB opens of the same file by the same DOS mode process is supported . The first open is passed through to the FSD. Subsequent opens are not seen by the FSD.

Any non-zero value returned by the FSD indicates that the open failed and the file is not open.

Note: This entry point's parameter list definition has changed from the 1.x IFS document. The OpenMode parameter has been widened from a unsigned short to a unsigned long. The upper word of the long is relevant only to a special SPOOLER FSD. For information about the upper word please contact the OS/2 Techinal Interface group for the OEMI document for the 2.0 API

FS_OPENPAGEFILE - Create paging file and handle

Purpose

Creates/opens the paging file for the Pager.

Calling Sequence

```
int far pascal FS_OPENPAGEFILE(pFlags, pcMaxReq, pName, psffsi, psffsd,
                               usOpenMode, usOpenFlag, usAttr, Reserved)

unsigned long far * pFlag;
unsigned long far * pcMaxReq;
char far * pName;
struct sffsi far * psffsi;
struct sffsd far * psffsi;
unsigned short usOpenMode;
unsigned short usOpenFlag;
unsigned short usAttr;
unsigned long Reserved;
```

Where

pFlag is a pointer to a flag double word for passing of information between the pager and the file system.

pFlag == 0x00000001 indicates first open of the page file.
pFlag == 0x00004000 indicates physical addresses are required in the page list .
pFlag == 0x00008000 indicates 16:16 virtual addresses are required in the page list.

All other values are reserved.

pcMaxReq is a pointer to a unsigned long where the FSD places the maximum request list length that can be managed by an enhanced strategy device driver.

pName is a pointer to the ASCIIZ path and filename of the paging file.

psffsi is a pointer to the file-system-independent portion of an open file instance.

psffsd is a pointer to the file-system-dependent portion of an open file instance.

usOpenMode indicates the desired sharing mode and access mode for the file handle.

See *OS/2 Version 2.0 Control Program Programming Reference* for a description of the OpenMode parameter for DosOpen.

usOpenFlag indicates the action taken when the file is present or absent.

See *OS/2 Version 2.0 Control Program Programming Reference* for a description of the usOpenFlag parameter for DosOpen.

usAttr are the OS/2 file attributes.

Reserved is a double word parameter reserved for use in the future.

Remarks

Enough information is provided for the FSD to perform a 'normal' open /create call.

Since a page file has special requirements about contiguity of its allocations, FS_OPENPAGEFILE must assure that any data sectors allocated are returned (Create call only). FS_ALLOCATEPAGESPACE will be called to handle file allocation .

If the FSD cannot support the FS_DOPAGEIO (usually due to an disk device driver which does not support the Extended strategy entry point), the FSD can return zero (0) for *pcMaxReq. This tells the kernel file system that it must emulate FS_DOPAGEIO.

The FSD can require either physical or virtual (16:16) addresses for subsequent calls to FS_DOPAGEIO. This allows an FSD to emulate FS_DOPAGEIO without having to worry about dealing with physical addresses.

For a detailed description of the Extended Strategy request interface please see the *OS/2 Version 2.0 Physical Device Driver Reference* .

FS_PATHINFO - Query/Set a File's Information

Purpose

Returns information for a specific path or file.

Calling Sequence

```
int far pascal FS_PATHINFO(flag, pcdfsi, pcdfsd, pName, iCurDirEnd, level,
                           pData, cbData)

unsigned short flag;
struct cdfsi far * pcdfsi;
struct cdfsd far * pcdfsd;
char far * pName;
unsigned short iCurDirEnd;
unsigned short level;
char far * pData;
unsigned short cbData;
```

Where

flag indicates retrieval or setting of information.

flag == 0x0000 indicates retrieving information
flag == 0x0001 indicates setting information on the media
flag == 0x0010 indicates that the information being set must be written-through onto the disk before returning. This bit is never set when retrieving information.

All other values are reserved.

pcdfsi is a pointer the file-system-independent working directory structure .

pcdfsd is a pointer to the file-system-dependent working directory structure .

pName is a pointer to the ASCIIZ name of the file or directory for which information is to be retrieved or set.

The FSD does not need to verify this pointer.

iCurDirEnd is the index of the end of the current directory in pName.

This is used to optimize FSD path processing. If iCurDirEnd == -1, there is no current directory relevant to the name text, that is a device.

level is the information level to be returned.

Level selects among a series of data structures to be returned or set.

pData is the address of the application data area.

Addressing of this data area is not validated by the kernel (see FSH_PROBEBUF). When retrieval (flag == 0) is specified, the FSD places the information into the buffer. When outputting information to a file (flag == 1), the FSD retrieves that data from the application buffer.

cbData is the length of the application data area.

For flag == 0, this is the length of the data the application wishes to retrieve. If there is not enough room for the entire level of data to be returned, the FSD returns a BUFFER OVERFLOW error. For flag == 1, this is the length of data to be applied to the file.

Remarks

See the descriptions of DosQPathInfo and DosSetPathInfo in the *OS/2 Version 2 .0 Control Program Programming Reference* for details on information levels.

The FSD will not be called for DosQPathInfo level 5.

FSDs that are case-preserving (like HPFS) can decide to accept level 7 requests. A level 7 DosQueryPathInfo request asks the FSD to fill the pData buffer with the case-preserved path and filename of the path/filename passed in pName. Routing of level 7 requests will be determined by the kernel by checking the LV7 bit in a FSD's attribute double word.

FS_PROCESSNAME - Allow FSD to modify name after OS/2 canor

Purpose

Allow an FSD to modify filename to its own specification after the OS/2 canonicalization process has completed.

Calling Sequence

```
int far pascal FS_PROCESSNAME(pNameBuf)

char far * pNameBuf;
```

Where

pNameBuf is a pointer to the ASCIIZ pathname.

The FSD should modify the pathname in place. The buffer is guaranteed to be the length of the maximum path. The FSD does not need to verify this pointer.

Remarks

The resulting name must be within the maximum path length returned by DosQSysInfo.

This routine allows the FSD to enforce a different naming convention than OS/ 2. For example, an FSD could remove blanks embedded in component names or return an error if it found such blanks. It is called after the OS/2 canonicalization process has succeeded. It is not called for FSH_CANONICALIZE.

This routine is called for all APIs that use pathnames.

This routine must return no error if the function is not supported.

FS_READ - Read from a File

Purpose

Read the specified number of bytes from a file to a buffer location.

Calling Sequence

```
int far pascal FS_READ(psffsi, psffsd, pData, pLen, IOflag)

struct sffsi far * psffsi;
struct sffsd far * psffsd;
char far * pData;
unsigned short far * pLen;
unsigned short IOflag;
```

Where

psffsi is a pointer to the file-system-independent portion of an open file instance.

sfi_position is the location within the file where the data is to be read from. The FSD should update the sfi_position field.

psffsd is a pointer to the file-system-dependent portion of an open file instance.

pData is the address of the application data area.

Addressing of this data area has not been validated by the kernel (see FSH_ PROBEBUF).

pLen is a pointer to the length of the application data area.

On input, this is the number of bytes to be read. On output, this is the number of bytes successfully read. If the application data area is smaller than the length, no transfer is to take place. The FSD will not be called for zero length reads. The FSD does not need to verify this pointer.

IOflag indicates information about the operation on the handle.

IOflag == 0x0010 indicates write-through
IOflag == 0x0020 indicates no-cache

Remarks

If read is successful and is a file, the FSD should set ST_SREAD and ST_PREAD to make the kernel time stamp the last modification time in the SFT.

Of the information passed in IOflag, the write-through bit is a mandatory bit in that any data written to the block device must be put out on the medium before the device driver returns. The no-cache bit, on the other hand, is an advisory bit that says whether the data being transferred is worth caching or not.

FS_RMDIR - Remove Subdirectory

Purpose

Removes a subdirectory from the specified disk.

Calling Sequence

```
int far pascal FS_RMDIR(pcdfsi, pcdfsd, pName, iCurDirEnd)

struct cdfsi far * pcdfsi;
struct cdfsd far * pcdfsd;
char far * pName;
unsigned short iCurDirEnd;
```

Where

pcdfsi is a pointer to the file-system-independent working directory structure.

pcdfsd is a pointer to the file-system-dependent working directory structure .

pName is a pointer to the ASCIIZ name of the directory to be removed.

The FSD does not need to verify this pointer.

iCurDirEnd is the index of the end of the current directory in pName.

This is used to optimize FSD path processing. If iCurDirEnd == -1, there is no directory relevant to the name text, that is a device.

Remarks

OS/2 assures that the directory being removed is not the current directory nor the parent of any current directory of any process.

The FSD should not remove any directory that has entries other than ' .' and '..' in it.

FS_SETSWAP - Notification of swap-file ownership

Purpose

Perform whatever actions are necessary to support the swapper.

Calling Sequence

```
int far pascal FS_SETSWAP(psffsi, psffsd)

struct sffsi far * psffsi;
struct sffsd far * psffsd;
```

Where

psffsi is a pointer to the file-system-independent portion of an open file instance of the swapper file.

psffsd is a pointer to the file-system-dependent portion of an open file instance.

Remarks

Swapping does not begin until this call returns successfully. This call is made during system initialization.

The FSD makes all segments that are relevant to swap-file I/O non-swappable (see FSH_FORCENOSWAP). This includes any data and code segments accessed during a read or write.

Any FSD that manages writeable media may be the swapper file system.

FS_SETSWAP may be called more than once for the same or different volumes or FSDs.

FS_SHUTDOWN - Shutdown file system

Purpose

Used to shutdown an FSD in preparation for power-off or IPL.

Calling Sequence

```
int far pascal FS_SHUTDOWN(type, reserved)

unsigned short type;
unsigned long reserved;
```

Where

type indicates what type of a shutdown operation to perform.

type == 0 indicates that the shutdown sequence is beginning. The kernel will not allow any new I/O calls to reach the FSD. The only exception will be I/ O to the swap file by the swap thread through the FS_READ and FS_WRITE entry points. The kernel will still allow any thread to call FS_COMMIT, FS_FLUSHBUF and FS_SHUTDOWN. The FSD should complete all pending calls that might generate disk corruption.

type == 1 indicates that the shutdown sequence is ending. An FS_COMMIT has been called on every SFT open on the FSD and following that an FS_FLUSHBUF on all volumes has been called. All final clean up activity must be completed before this call returns.

reserved reserved for future expansion.

Remarks

From the perspective of an FSD, the shutdown sequence looks like this:

First, the system will call the FSD's FS_SHUTDOWN entry with type == 0 . This notifies the FSD that the system will begin committing SFTs in preparation for system power off. The kernel will not allow any new IO calls to the FSD once it receives this first call, except from the swapper thread. The swapper thread will continue to call the FS_READ and FS_WRITE entry points to read and write the swap file. The swapper thread will not attempt to grow or shrink the swap file nor should the FSD reallocate it. The kernel will continue to allow FS_ COMMIT and FS_FLUSHBUF calls from any thread. This call should not return from the FSD until disk data modifying calls have completed to insure that a thread already inside the FSD does not wake and change disk data.

After the first FS_SHUTDOWN call returns, the kernel will start committing SFTs. The FSD will see a commit for every SFT associated with it. During these FS_COMMIT calls, the FSD must flush any data associated with these SFTs to disk . The FSD must not allow any FS_COMMIT or FS_FLUSHBUF call to block permanently .

Once all of the SFTs associated with the FSD have been committed, FS_SHUTDOWN will be called with type == 1. This will tell the FSD to flush all buffers to disk. From this point, the FSD must not buffer any data destined for disk. Reads and writes to the swap file will continue, but the allocation of the swap file will not change. Once this call has completed, no file system corruption should occur if power is shut off.

FS_VERIFYUNCNAME - Verify UNC server ownership

Purpose

Used to poll installed UNC FSDs to determine server ownership.

Calling Sequence

```
int far pascal FS_VERIFYUNCNAME(flag, pName)

unsigned short flag;
char far * pName;
```

Where

flag indicates which 'pass' of the poll the FSD is being called.

flag == 0x0000 indicates that this is a pass 1 poll.
flag == 0x0001 indicates that this is a pass 2 poll.

pName is a pointer to the ASCIIZ name of the server in UNC format.

The FSD does not need to verify this pointer.

Remarks

What the kernel expects from UNC FSDs for this entry point:

For pass 1, the FSD will be called and passed a pointer to the UNC server name. It is to respond immediately if it recognizes (manages) the server with a NO_ERROR return code. This pass expects the that the FSD will be keeping tables in memory that contain the UNC names of the servers it is currently servicing . If the UNC name cannot be validated immediately, the FSD should respond with an error (non-zero) return code. The FSD SHOULD NOT send messages in an attempt to validate the server name.

For pass 2, the FSD is permitted to do whatever is reasonable, including sending LAN 'are you there' messages, to determine if they are able to service the request for UNC server.

FS_WRITE - Write to a file

Purpose

Write the specified number of bytes to a file from a buffer location.

Calling Sequence

```
int far pascal FS_WRITE(psffsi, psffsd, pDat, pLen, IOflag)

struct sffsi far * psffsi;
struct sffsd far * psffsd;
char far * pData;
unsigned short far * pLen;
unsigned short IOflag;
```

Where

psffsi is a pointer to the file-system-independent portion of an open file instance.

sfi_position is the location within the file where the data is to be written to. The FSD should update the sfi_position and sfi_size fields.

psffsd is a pointer to the file-system-dependent portion of an open file instance.

pData is the address of the application data area.

Addressing of this data area is not validated by the kernel (see FSH_PROBEBUF).

pLen is a pointer to the length of the application data area.

On input, this is the number of bytes that are to be written. On output, this is the number of bytes successfully written. If the application data area is smaller than the length, no transfer is to take place. The FSD does not need to verify this pointer.

IOflag indicates information about the operation on the handle.

IOflag == 0x0010 indicates write-through
IOflag == 0x0020 indicates no-cache

Remarks

If write is successful and is a file, the FSD should set ST_SWRITE and ST_ PWRITE to make the kernel time stamp the last modification time in the SFT.

The FSD should return an error if an attempt is made to write beyond the end with a direct access device handle.

Of the information passed in IOflag, the write-through bit is a mandatory bit in that any data written to the block device must be put out on the medium before the device driver returns. The no-cache bit, on the other hand, is an advisory bit that says whether the data being transferred is worth caching or not.

FS Helper Functions

The following table summarizes the routines that make up the File System Helper interface between FSDs and the kernel.

FS Helper Routine	Description
FSH_ADDSHARE	Add a name to the sharing set
FSH_BUFSTATE	REMOVED in OS/2 Version 2.0
FSH_CALLDRIVER	Call Device Driver's Extended Strategy entry point
FSH_CANONICALIZE	Convert pathname to canonical form
FSH_CHECKEANAME	Check EA name validity
FSH_CRITERROR	Signal a hard error to the daemon
FSH_DEVIOCTL	Send IOCTL request to device driver
FSH_DOVOLIO	Volume-based sector-oriented transfer
FSH_DOVOLIO2	Send volume-based IOCTL request to device driver.
FSH_FINDCHAR	Find first occurrence of char in string
FSH_FINDDUPHVPB	Locates equivalent hVPBs
FSH_FLUSHBUF	REMOVED in OS/2 Version 2.0
FSH_FORCENOSWAP	Force segments permanently into memory
FSH_GETBUF	REMOVED in OS/2 Version 2.0
FSH_GETFIRSTOVERLAPB	REMOVED in OS/2 Version 2.0
FSH_GETPRIORITY	Get current thread's I/O priority
FSH_GETVOLPARM	Get VPB data from VPB handle
FSH_INTERR	Signal an internal error
FSH_IOBOOST	Gives the current thread an I/O priority boost
FSH_IOSEMCLEAR	Clear an I/O-event semaphore
FSH_ISCURDIRPREFIX	Test for a prefix of a current directory
FSH_LOADCHAR	Load character from a string
FSH_NAMEFROMSFN	Get the full path name from an SFN
FSH_PREVCHAR	Move backward in string
FSH_PROBEBUF	User address validity check
FSH_QSYSINFO	Query system information
FSH_REGISTERPERFCTRS	Register a FSD with PERFVIEW

FSH_RELEASEBUF	REMOVED in OS/2 Version 2.0
FSH_REMOVESHARE	Remove a name from the sharing set
FSH_SEGALLOC	Allocate a GDT or LDT segment
FSH_SEGFREE	Release a GDT or LDT segment
FSH_SEGREALLOC	Change segment size
FSH_SEMCLEAR	Clear a semaphore
FSH_SEMREQUEST	Request a semaphore
FSH_SEMSET	Set a semaphore
FSH_SEMSETWAIT	Set a semaphore and wait for clear
FSH_SEMWAIT	Wait for clear
FSH_SETVOLUME	force a volume to be mounted on the drive
FSH_STORECHAR	Store character into string
FSH_UPPERCASE	Uppercase ASCIIZ string
FSH_WILDMATCH	Match using OS/2 wildcards
FSH_YIELD	Yield CPU to higher priority threads

FSDs are loaded as dynamic link libraries and may import services provided by the kernel. These services can be called directly by the file system, passing the relevant parameters.

No validation of input parameters is done unless otherwise specified. The FSD calls FSH_PROBEBUF, where appropriate, before calling the FS help routine.

When any service returns an error code, the FSD must return to the caller as soon as possible and return the specific error code from the helper to the FS router .

There are many deadlocks that may occur as a result of operations issued by FSDs. OS/2 provides no means whereby deadlocks between file systems and applications can be detected.

FSH_ADDSHARE - Add a name to the share set

Purpose

This function adds a name to the currently active sharing set.

Calling Sequence

```
int far pascal FSH_ADDSHARE(pName, mode, hVPB, phShare)

char far * pName;
unsigned short mode;
unsigned short hVPB;
unsigned long far * phShare;
```

Where

pName is a pointer to the ASCIIZ name to be added into the share set.

The name must be in canonical form: no '.' or '. .' components, uppercase, no meta characters, and full path name specified .

mode is the sharing mode and access mode as defined in the DosOpen API. All other bits (direct open, write-through, etc.) must be zero.

hVPB is the handle to the volume where the named object is presumed to exist .

phShare is the pointer to the location where a share handle is stored. This handle may be passed to FSH_REMOVESHARE.

pName is a pointer to the ASCIIZ name to be added into the share set.

The name must be in canonical form: no '.' or '. .' components, uppercase, no meta characters, and full path name specified .

mode is the sharing mode and access mode as defined in the DosOpen API. All other bits (direct open, write-through, etc.) must be zero.

hVPB is the handle to the volume where the named object is presumed to exist .

phShare is the pointer to the location where a share handle is stored. This handle may be passed to FSH_REMOVESHARE.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_SHARING_VIOLATION
the file is open with a conflicting sharing/access mode.

oERROR_TOO_MANY_OPEN_FILES
there are too many files open at the present time.

oERROR_SHARING_BUFFER_EXCEEDED
there is not enough memory to hold sharing information.

oERROR_INVALID_PARAMETER
invalid bits in mode.

oERROR_FILENAME_EXCED_RANGE
path name is too long.

Remarks

Do not call FSH_ADDSHARE for character devices.

FSH_ADDSHARE may block.

Note: OS/2 does not validate input parameters. Therefore, an FSD should call FSH_PROBEBUF where appropriate.

FSH_CALLDRIVER - Call Device Driver's Extended Strategy entry

Purpose

This routine allows FSDs to call a device driver's Extended Strategy entry point.

Calling Sequence

```
int far pascal FSH_CALLDRIVER(pPkt, hVPB, fControl)

void far * pPkt;
unsigned short hVPB;
unsigned short fControl;
```

Where

pPkt is a pointer to device driver Extended strategy request packet.

hVPB is the volume handle for the source of I/O.

fControl is the bit mask of pop-up control actions:

Bit 0 off indicates volume change pop-up desired
Bit 0 on indicates no volume change pop-up

All other bits are reserved and must be zero.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_VOLUME_CHANGED
is an indication that removable media volume change has occurred.

oERROR_INVALID_PARAMETER
the fControl flag word has reserved bits on.

Remarks

This routine should be called for any Extended strategy requests going to a drive that has removable media.

For a detailed description of the Extended Strategy request interface please see the *OS/2 Version 2.0 Physical Device Driver Reference*

FSH_CALLDRIVER may block.

Note: OS/2 does not validate input parameters. Therefore, an FSD should call FSH_PROBEBUF where appropriate.

FSH_CANONICALIZE - Convert a path name to a canonical form

Purpose

This function converts a path name to a canonical form by processing ' .'s and '..'s, uppercasing, and prepending the current directory to non-absolute paths.

Calling Sequence

```
int far pascal FSH_CANONICALIZE(pPathName, cbPathBuf, pPathBuf, pFlags)

char far * pPathName;
unsigned short cbPathBuf;
char far * pPathBuf;
unsigned short far * pFlags;
```

Where

pPathName is a pointer to the ASCIIZ path name to be canonicalized.

cbPathBuf is the length of path name buffer.

pPathBuf is the pointer to the buffer into which to copy the canonicalized path.

pFlags is the pointer to flags returned to the FSD.

Flags == 0x0040 indicates a non-8.3 filename format. All other values are reserved.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_PATH_NOT_FOUND
is an invalid path name-too many '..'s

oERROR_BUFFER_OVERFLOW
the path name is too long.

Remarks

This routine processes DBCS characters properly.

The FSD is responsible for verifying the string pointers and checking for segment boundaries.

FSH_CANONICALIZE should be called for names passed into the FSD raw data packets. For example, names passed to FS_FSCTL in the parameter area should be passed to FSH_CANONICALIZE. This routine does not need to be called for explicit names passed to the FSD, that is, the name passed to FS_OPENCREATE.

If the canonicalized name is being created as a file or directory, the non-8 .3 attribute in the directory entry should be set according to the value returned in pFlags.

Note: OS/2 does not validate input parameters. Therefore, an FSD should call FSH_PROBEBUF where appropriate.

FSH_CHECKEANAME - Check for valid EA name

Purpose

Check extended attribute name validity.

Calling Sequence

```
int far pascal FSH_CHECKEANAME(iLevel, cbEAName, szEAName)

unsigned short iLevel;
unsigned long cbEAName;
char far * szEAName;
```

Where

iLevel is the extended attributes name checking level.

iLevel = 0x0001 indicates OS/2 Version 2.0 name checking.

cbEAName is the length of the extended attribute name, not including terminating NUL.

szEAName is the extended attribute name to check for validity.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_INVALID_NAME
pathname contains invalid or wildcard characters, or is too long.

oERROR_INVALID_PARAMETER
invalid level.

Remarks

This routine processes DBCS characters properly.

The set of invalid characters for EA names is the same as that for filenames . In OS/2 Version 2.0, the maximum length of an EA name, not including the terminating NUL, is 255 bytes. The minimum length is 1 byte.

The FSD is responsible for verifying the string pointers and checking for segment boundaries.

FSH_CHECKEANAME should be called for extended attribute names passed to the FSD.

Note: OS/2 does not validate input parameters. Therefore, an FSD should call FSH_PROBEBUF where appropriate.

FSH_CRITERROR - Signal hard error to daemon

Purpose

This function signals a hard error to the daemon.

Calling Sequence

```
int far pascal FSH_CRITERROR(cbMessage, pMessage, nSubs, pSubs, fAllowed)

unsigned short cbMessage;
char far * pMessage;
unsigned short nSubs;
char far * pSubs;
unsigned short fAllowed;
```

Where

cbMessage is the length of the message template.

pMessage is the pointer to the message template.

This may contain replaceable parameters in the format used by the message retriever.

nSubs is the number of replaceable parameters.

pSubs is the pointer to the replacement text.

The replacement text is a packed set of ASCIIZ strings.

fAllowed is the bit mask of allowed actions:

Bit 0x0001 on indicates FAIL allowed
Bit 0x0002 on indicates ABORT allowed
Bit 0x0004 on indicates RETRY allowed
Bit 0x0008 on indicates IGNORE allowed
Bit 0x0010 on indicates ACKNOWLEDGE only allowed.

All other bits are reserved and must be zero. If bit 0x0010 is set, and any or some of bits 0x0001 to 0x0008 are also set, bit 0x0010 will be ignored.

Returns

This function returns the action to be taken:

0x0000 - ignore
0x0001 - retry
0x0003 - fail
0x0004 - continue

Remarks

If the user responds with an action that is not allowed, it is treated as FAIL. If FAIL is not allowed, it is treated as ABORT. ABORT is always allowed.

When ABORT is the final action, OS/2 does not return this as an indicator; it returns a FAIL. The actual ABORT operation is generated when this thread of execution is about to return to user code.

The maximum length of the template is 128 bytes, including the NUL. The maximum length of the message with text substitutions is 512 bytes. The maximum number of substitutions is 9.

If any action other than retry is selected for a given hard error popup, then any subsequent popups (within the same API call) will be automatically failed; a popup will not be done. This means that (except for retries) there can be at most one hard error popup per call to the FSD. And, if the kernel generates a popup, then the FSD cannot create one.

FSH_CRITERROR will fail automatically if the user application has set autofail, or if a previous hard error has occurred.

FSH_CRITERROR may block.

Note: OS/2 does not validate input parameters. Therefore, an FSD should call FSH_PROBEBUF where appropriate.

FSH_DEVIOCTL - Send IOCTL request to device driver

Purpose

This function sends an IOCTL request to a device driver.

Calling Sequence

```
int far pascal FSH_DEVIOCTL(flag, hDev, sfn, cat, func, pParm, cbParm, pData,
                            cbData)

unsigned short flag;
unsigned long hDev;
unsigned short sfn;
unsigned short cat;
unsigned short func;
char far * pParm;
unsigned short cbParm;
char far * pData;
unsigned short cbData;
```

Where

flag indicates whether the FSD initiated the call or not.

IOflag == 0x0000 indicates that the FSD is just passing user pointers on to the helper.
IOflag == 0x0001 indicates that the FSD initiated the DevIOCtl call as opposed to passing a DevIOCtl that it had received.

All other bits are reserved.

hDev is the device handle obtained from VPB

sfn is the system file number from open instance that caused the FSH_DEVIOCTL call.

This field should be passed unchanged from the sfi_selfsfn field. If no open instance corresponds to this call, this field should be set to 0xFFFF.

cat is the category of IOCTL to perform.

func is the function within the category of IOCTL.

pParm is the long address to the parameter area.

cbParm is the length of the parameter area.

pData is the long address to the data area.

cbData is the length of the data area.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_INVALID_FUNCTION
indicates the function supplied is incompatible with the category and device handle supplied.

oERROR_INVALID_CATEGORY
indicates the category supplied is incompatible with the function and device handle supplied.

oDevice driver error code

Remarks

The only category currently supported for this call is 8, which is for the logical disk. FSDs call FSH_DEVIOCTL to control device driver operation independently from I/O operations. This is typically in filtering DosDevIOCtl requests when passing the request on to the device driver.

An FSD needs to be careful of pointers to buffers that are passed to it from FS_IOCTL, and what it passes to FSH_DEVIOCTL. It is possible that such pointers may be real mode pointers if the call was made from the DOS mode. In any case, the FSD must indicate whether it initiated the DevIOCtl call, in which case the kernel can assume that the pointers are all protect mode pointers, or if it is passing user pointers on to the FSH_DEVIOCTL call, in which case the mode of the pointers will depend on whether this is the DOS mode or not. An important thing to note is that the FSD must not mix user pointers with its own pointers when using this helper.

FSH_DEVIOCTL may block.

Note: OS/2 does not validate input parameters. Therefore, an FSD should call FSH_PROBEBUF where appropriate.

FSH_DOVOLIO - Transfer volume-based sector-oriented I/O

Purpose

This function performs I/O to the specified volume.

Calling Sequence

```
int far pascal FSH_DOVOLIO(operation, fAllowed, hVPB, pData, pcSec, iSec)

unsigned short operation;
unsigned short fAllowed;
unsigned short hVPB;
char far * pData;
unsigned short far * pcSec;
unsigned long iSec;
```

Where

operation is the bit mask indicating read/read-bypass/write/write-bypass, and verify- after-write/write-through and no-cache operation to be performed.

Bit 0x0001 off indicates read.
Bit 0x0001 on indicates write.
Bit 0x0002 off indicates no cache bypass.
Bit 0x0002 on indicates cache bypass.
Bit 0x0004 off indicates no verify-after-write operation.
Bit 0x0004 on indicates verify-after-write operation.
Bit 0x0008 off indicates errors signaled to the hard error daemon.
Bit 0x0008 on indicates hard errors will be returned directly.
Bit 0x0010 off indicates I/O is not write-through.
Bit 0x0010 on indicates I/O is write-through.
Bit 0x0020 off indicates data for this I/O should probably be cached.
Bit 0x0020 on indicates data for this I/O should probably not be cached.
All other bits are reserved and must be zero.

The difference between the cache bypass and the no cache bits is in the type of request packet that the device driver will see. With cache bypass, it will get a packet with command code 24, 25, or 26. With no cache, it will get the extended packets for command codes 4, 8, or 9. The advantage of the latter is that the write-through bit can also be sent to the device driver in the same packet, improving the functionality at the level of the device driver.

fAllowed is a bit mask indicating allowed actions:

Bit 0x0001 on indicates FAIL allowed
Bit 0x0002 on indicates ABORT allowed
Bit 0x0004 on indicates RETRY allowed
Bit 0x0008 on indicates IGNORE allowed
Bit 0x0010 on indicates ACKNOWLEDGE only allowed
If this bit is set, none of the other bits may be set.

All other bits are reserved and must be set to zero.

hVPB is the volume handle for the source of I/O.

pData is the long address of the user transfer area.

pcSec is the pointer to the number of sectors to be transferred.

On return, this is the number of sectors successfully transferred.

iSec is the sector number of the first sector of the transfer.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_PROTECTION_VIOLATION
indicates the supplied address/length is invalid.

oERROR_UNCERTAIN_MEDIA
indicates the device driver can no longer reliably tell if the media has been changed.

This occurs only within the context of an FS_MOUNT call.

oERROR_TRANSFER_TOO_LONG
indicates the transfer is too long for the device.

oDevice-driver/device-manager errors listed /DDERR/

Remarks

This function formats a device driver request packet for the requested I/O, locks the data transfer region, calls the device driver, and reports any errors to the hard error daemon before returning to the FSD. Any retries indicated by the hard error daemon or actions indicated by DosError are done within the call to FSH_ DOVOLIO.

FSH_DOVOLIO may be used at all times within the FSD. When called within the scope of a FS_MOUNT call, it applies to the volume in the drive without regard to which volume it may be. However, since volume recognition is not complete until the FSD returns to the FS_MOUNT call, the FSD must be careful when an ERROR_ UNCERTAIN_MEDIA is returned. This indicates the media has gone uncertain while we are trying to identify the media in the drive. This may indicate the volume that the FSD was trying to recognize was removed. In that case, the FSD must release any resources attached to the hVPB passed in the FS_MOUNT call and return ERROR_UNCERTAIN_ MEDIA to the FS_MOUNT call. This will direct the volume tracking logic to restart the mount process.

OS/2 will validate the user transfer area for proper access and size and will lock the segment.

Verify-after-write specified on a read is ignored.

On 80386 processors, FSH_DOVOLIO will take care of memory contiguity requirements of device drivers. It is advisable, therefore, that FSDs use FSH_DOVOLIO instead of calling device drivers directly. This will improve performance of FSDs running on 80386 processors.

FSH_DOVOLIO may block.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_DOVOLIO2 - Send volume-based IOCTL request to device dri

Purpose

This function is an FSD call that controls device driver operation independently from I/O operations.

Calling Sequence

```
int far pascal FSH_DOVOLIO2(hDev, sfn, cat, func, pParm, cbParm, pData,
                            cbData)

unsigned long hDev;
unsigned short sfn;
unsigned short cat;
unsigned short func;
char far * pParm;
unsigned short cbParm;
char far * pData;
unsigned short cbData;
```

Where

hDev is the device handle obtained from VPB

sfn is the system file number from the open instance that caused the FSH_ DOVOLIO2 call.

This field should be passed unchanged from the sfi-selfsfn field. It no open instance corresponds to this call, this field should be set to 0xFFFF.

cat is the category of IOCTL to perform.

func is the function within the category of IOCTL.

pParm is the long address to the parameter area.

cbParm is the length of the parameter area.

pData is the long address to the data area.

cbData is the length of the data area.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_INVALID_FUNCTION
indicates the function supplied is incompatible with the category and the device handle supplied.

oERROR_INVALID_CATEGORY
indicates the category supplied is incompatible with the function and the device handle supplied.

oDevice-driver/device-manager errors listed /DDERR/

Remarks

This routine supports volume management for IOCTL operations. Any errors are reported to the hard error daemon before returning to the FSD. Any retries indicated by the hard error daemon or actions indicated by DosError are done within the call to FSH_DOVOLIO2.

The purpose of this routine is to enable volume tracking with IOCTLs. It is not available at the API level.

FSH_DOVOLIO2 may block.

System does normal volume checking for this request.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_FINDCHAR - Find first occurrence of character in string

Purpose

This function provides the mechanism to find the first occurrence of any one of a set of characters in an ASCIIZ string, taking into account DBCS considerations.

Calling Sequence

```
int far pascal FSH_FINDCHAR(nChars, pChars, ppStr)

unsigned short nChars;
char far * pChars;
char far * far * ppStr;
```

Where

nChars is the number of characters in the search list.

pChars is the array of characters to search for. These cannot be DBCS characters.

The NUL character cannot be searched for.

ppSTR is the pointer to the character pointer where the search is to begin . This pointer is updated upon return to point to the character found. This must be an ASCIIZ string.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_CHAR_NOT_FOUND
indicates none of the characters were found.

Remarks

The search will continue until a matching character is found or the end of the string is found.

The FSD is responsible for verifying the string pointers and checking for segment boundaries.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_FINDDUPHVPB - Locate equivalent hVPB

Purpose

This function provides the mechanism to identify a previous instance of a volume during the FS_MOUNT process.

Calling Sequence

```
int far pascal FSH_FINDDUPHVPB(hVPB, phVPB)

unsigned short hVPB;
unsigned short far * phVPB;
```

Where

hVPB is the handle to the volume to be found.

phVPB is the pointer to where the handle of matching volume will be stored .

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_NO_ITEMS
indicates there is no matching hVPB.

Remarks

When OS/2 is recognizing a volume, it calls the FSD to mount the volume. At this point, the FSD may elect to allocate storage and buffer data for that volume. The mount process will allocate a new VPB whenever the media becomes uncertain, that is, when the device driver recognizes it can no longer be certain the media is unchanged. This VPB cannot be collapsed with a previously allocated VPB, because of a reinsertion of media, until the FS_MOUNT call returns. The previous VPB, however, may have some cached data that must be updated from the media (the media may have been written while it was removed) FSH_FINDDUPHVPB allows the FSD to find this previous occurrence of the volume in order to update the cached information for the old VPB. Remember the newly created VPB will be unmounted if there is another, older VPB for that volume.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_FORCENOSWAP - Force segments permanently into memor

Purpose

This function permanently forces segments into memory.

Calling Sequence

```
int far pascal FSH_FORCENOSWAP(sel)

unsigned short sel;
```

Where

sel is the selector that is to be made non-swappable.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_INVALID_ACCESS
indicates the selector is invalid.

oERROR_INVALID_DENIED
indicates the selector is invalid or the sector belongs to another process.

oERROR_DIRECT_ACCESS_HANDLE
indicates the handle does not refer to a segment.

oERROR_NOT_ENOUGH_MEMORY
indicates there is not enough physical memory to make a segment nonswappable .

oERROR_SWAP_TABLE_FULL
indicates the attempt to grow the swap file failed.

oERROR_SWAP_FILE_FULL
indicates the attempt to grow the swap file failed.

oERROR_PMM_INSUFFICIENT_MEMORY
indicates the attempt to grow the swap file failed.

Remarks

An FSD may call FSH_FORCENOSWAP to force segments to be loaded into memory and marked non-swappable. All segments both in the load image of the FSD and those allocated via FSH_SEGALLOC are eligible to be marked. There is no way to undo the effect of FSH_FORCENOSWAP.

If an FSD is notified it is managing the swapping media, it should make this call for the necessary segments.

An FSD should be prepared to see multiple swapping files on more than one volume in 80386 processors and in future releases of OS/2.

FSH_FORCENOSWAP may block.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_GETPRIORITY - Get current thread's I/O priority

Purpose

This function allows an FSD to retrieve the I/O priority of the current thread.

Calling Sequence

```
int far pascal FSH_GETPRIORITY(void)
```

Returns

This function returns the I/O of the current thread:

0x0000 - background
0x1111 - foreground

Remarks

FSH_GETPRIORITY will not block.

FSH_GETVOLPARM - Get VPB data from VPB handle

Purpose

This function allows an FSD to retrieve file-system-independent and file- system-dependent data from a VPB. Since the FS router passes in a VPB handle, individual FSDs need to map the handle into pointers to the relevant portions.

Calling Sequence

```
void far pascal FSH_GETVOLPARM(hVPB, ppVPBfsi, ppVPBfsd)

unsigned short hVPB;
struc vpfsi far * far * ppVPBfsi;
struc vpfsd far * far * ppVPBfsd;
```

Where

hVPB is the volume handle of interest.

ppVPBfsi indicates the location where the pointer to file-system-independent data is stored.

ppVPBfsd indicates the location where the pointer to file-system-dependent data is stored.

Returns

There are no error returns.

Remarks

FSH_GETVOLPARM will not block.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_INTERR - Signal an internal error

Purpose

This function signals an internal error.

Calling Sequence

```
void far pascal FSH_INTERR(pMsg, cbMsg)

char far * pMsg;
unsigned short cbMsg;
```

Where

pMsg is a pointer to the message text.

cbMsg is the length of the message text.

Returns

There are no error returns.

Remarks

For reliability, if an FSD detects an internal inconsistency during normal operation, the FSD shuts down the system as a whole. This is the safest thing to do since it is not clear if the system as a whole is in a state that allows normal execution to continue.

When an FSD calls FSH_INTERR, the address of the caller and the supplied message is displayed on the console. The system then halts.

The code used to display the message is primitive. The message should contain ASCII characters in the range 0x20-0x7E, optionally with 0x0D and 0x0A to break the text into multiple lines.

The FSD must preface all such messages with the name of the file system.

The maximum message length is 512 characters. Messages longer than this are truncated.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_IOBOOST - Gives the current thread an I/O priority boost

Purpose

This function allows an FSD to boost the current thread's I/O priority after a I/O request.

Calling Sequence

```
void far pascal FSH_IOBOOST(void)
```

Returns

There are no error returns.

Remarks

FSH_IOBOOST will not block.

FSH_IOSEMCLEAR - Clear an I/O event semaphore

Purpose

This function allows an FSD to clear the I/O event semaphore that is a part of the Extended Strategy request packet.

Calling Sequence

```
int far pascal FSH_IOSEMCLEAR(pSem)
```

Where

pSem is the handle to the I/O event semaphore.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_EXCL_ALREADY_OWNED
the exclusive semaphore is already owned.

oERROR_PROTECTION_VIOLATION
the semaphore is inaccessible.

Remarks

FSH_IOSEMCLEAR may block.

For a detailed description of the Extended Strategy request interface, please see the *OS/2 Version 2.0 Physical Device Driver Reference*

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_ISCURDIRPREFIX - Test for a prefix of a current directory

Purpose

This function allows FSDs to disallow any modification of any directory that is either a current directory of some process or the parent of any current directory of some process. This is necessary because the kernel manages the text of each current directory for each process.

Calling Sequence

```
int far pascal FSH_ISCURDIRPREFIX(pName)

char far * pMsg;
```

Where

pName is a pointer to the path name.

The name must be in canonical form, that is, no '.' or ' ..' components, uppercase, no meta characters, and full path name specified.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_CURRENT_DIRECTORY
the specified path is a prefix of or is equal to the current directory of some process.

If the current directory is the root and the path name is `d:\ `, ERROR_CURRENT_DIRECTORY will be returned.

Remarks

FSH_ISCURDIRPREFIX takes the supplied path name, enumerates all current directories in use, and tests to see if the specified path name is a prefix or is equal to some current directory.

FSH_ISCURDIRPREFIX may block.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_LOADCHAR - Load a character from a string

Purpose

This function provides the mechanism for loading a character from a string, taking into account DBCS considerations.

Calling Sequence

```
void far pascal FSH_LOADCHAR(ppStr, pChar)

char far * far * ppStr;
unsigned short far * pChar;
```

Where

ppStr is a pointer to the character pointer of a string.

The character at this location will be retrieved and this pointer will be updated.

pChar is a pointer to the character returned.

If character is non-DBCS, the first byte will be the character and the second byte will be zero.

Returns

There are no error returns.

Remarks

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_NAMEFROMSFN - Get the full path name from an SFN.

Purpose

This call allows an FSD to retrieve the full path name for an object to which an SFN refers.

Calling Sequence

```
int far pascal FSH_NAMEFROMSFN(sfn, pName, pcbName)

unsigned short sfn;
char far * pName;
unsigned short far * pcbName;
```

Where

sfn is the system file number of a file instance, obtained from the sfi_ selfsfn field of the file-system-independent part of the SFT for the object.

pName is the location of where the returned full path name is to be stored .

pcbName is the location of where the FSD places the size of the buffer pointed to by pName. On return, the kernel will fill this in with the length of the path name. This length does not include the terminating null character. The size of the buffer should be long enough to hold the full path name, or else an error will be returned.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_INVALID_HANDLE
the SFN is invalid.

oERROR_BUFFER_OVERFLOW
the buffer is too short for the returned path.

Remarks

FSH_NAMEFROMSFN will not block.

Note: OS/2 does not validate input parameters; the FSD should call FSH_ PROBEBUF where appropriate.

FSH_PREVCHAR - Decrement a character pointer

Purpose

This function provides the mechanism for decrementing a character point, taking into account DBCS considerations.

Calling Sequence

```
void far pascal FSH_PREVCHAR(pBeg, ppStr)

char far * pBeg;
char far * far * ppStr;
```

Where

pBeg is a pointer to the beginning of a string.

ppStr is a pointer to the character pointer of a string.

The value is decremented appropriately upon return. If it is at the beginning of a string, the pointer is not decremented. If it points to the second byte of a DBCS character, it will be decremented to point to the first byte of the character.

Returns

There are no error returns.

Remarks

The FSD is responsible for verifying the string pointer and checking for segment boundaries.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_PROBEBUF - Check user address validity

Purpose

This function provides the mechanism for performing validity checks on arbitrary pointers to data that users may pass in.

Note: FSDs must check on these pointers before using them.

Calling Sequence

```
int far pascal FSH_PROBEBUF(operation, pdata, cbData)

unsigned short operation;
char far * pData;
unsigned short cbData;
```

Where

operation indicates whether read or write access is desired.

operation == 0 indicates read access is to be checked.
operation == 1 indicates write access is to be checked.

All other values are reserved.

pData is the starting address of user data.

cbData is the length of user data. If cbData is 0, a length of 64K is indicated.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_PROTECTION_VIOLATION
indicates access to the indicated memory region is illegal (access to the data is inappropriate or the user transfer region is partially or completely inaccessible).

Remarks

Because users may pass in arbitrary pointers to data, FSDs must perform validity checks on these pointers before using them. Because OS/2 is multi-threaded, the addressability of data returned by FSH_PROBEBUF is only valid until the FSD blocks. Blocking, either explicitly or implicitly allows other threads to run, possibly invalidating a user segment. FSH_PROBEBUF must, therefore, be reapplied after every block.

FSH_PROBEBUF provides a convenient method to assure a user transfer address is valid and present in memory. Upon successful return, the user address may be treated as a far pointer and accessed up to the specified length without either blocking or faulting. This is guaranteed until the FSD returns or until the next block.

If FSH_PROBEBUF detects a protection violation, the process is terminated as soon as possible. The OS/2 kernel kills the process once it has exited from the FSD.

On 80386 processors, FSH_PROBEBUF ensures all touched pages are physically present in memory so the FSD will not suffer an implicit block due to a page fault. However, FSH_PROBEBUF does NOT guarantee the pages will be physically contiguous in memory because FSDs are not expected to do DMA.

FSH_PROBEBUF may block.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_QSYSINFO - Query system information

Purpose

This function queries the system about dynamic system variables and static system variables not returned by DosQSysInfo.

Calling Sequence

```
int far pascal FSH_QSYSINFO(index, pData, cbData)

unsigned short index;
char far * pData;
unsigned short cbData;
```

Where

index is the variable to return.

index == 1 indicates maximum sector size.

index == 2 indicates process identity. The data returned will be as follows:

```
struct {
    unsigned short PID;
    unsigned short UID;
    unsigned short PDB;
}
```

index == 3 indicates absolute thread number for the current thread. This will be returned in an unsigned short field.

index == 4 indicates verify on write flag for the process. This will be returned in an unsigned char (byte) field. Zero means verify is off, non-zero means it is on.

pData is the long address to the data area.

cbData is the length of the data area.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_INVALID_PARAMETER
the index is invalid.

oERROR_BUFFER_OVERFLOW
the specified buffer is too short for the returned data.

Remarks

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_REGISTERPERFCTRS - Register a FSD with PERFVIEW

Purpose

This function allows the FSD to register with the PERFVIEW product. The FSD passes pointers to its counter data and text blocks.

Calling Sequence

```
int far pascal FSH_REGISTERPERFCTRS(pDataBlk, pTextBlk, fsFlags)

void far * pDataBlk;
void far * pTextBlk;
unsigned short fsFlags;
```

Where

pDataBlk is a pointer to the data block where the actual counters reside.

pTextBlk is a pointer to the block that contains instance and name information about counters in the associated DataBlk.

fsFlags indicates what type of addressing is going to be used.

Bit 0 off indicates 16:16 pointers
Bit 0 on indicates 0:32 pointers

All other bits are reserved and must be zero.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_INVALID_PARAMETER
the flag word is invalid.

oERROR_PVW_INVALID_COUNTER_BLK
the specified buffer is not in the correct PERFVIEW data block format

oERROR_PVW_INVALID_TEXT_BLK

the specified buffer is not in the correct PERFVIEW text block format

Remarks

For a detailed description of the PERFVIEW interface and its associated data structures please see the OS/2 Version 2.0 PERFVIEW OEMI Document.

FSH_REGISTERPERFCTRS may block.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_REMOVESHARE - Remove a shared entry

Purpose

This function removes a previously-entered name from the sharing set.

Calling Sequence

```
void far pascal FSH_REMOVESHARE(hShare)

unsigned long hShare;
```

Where

hShare is a share handle returned by a prior call to FSH_ADDSHARE.

Returns

There are no error returns.

Remarks

When a call to FSH_REMOVESHARE has been issued, the share handle is no longer valid.

FSH_REMOVESHARE may block.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_SEGALLOC - Allocate a GDT or LDT segment

Purpose

This function allocates a GDT or LDT selector. The selector will have read/write access. An FSD may call this function.

Calling Sequence

```
int far pascal FSH_SEGALLOC(flags, cbSeg, pSel)

unsigned short flags;
unsigned long cbSeg;
unsigned short far * pSel;
```

Where

flags indicate GDT/LDT, protection ring, swappable/non-swappable.

Bit 0x0001 off indicates GDT selector returned.
Bit 0x0001 on indicates LDT selector returned.
Bit 0x0002 off indicates non-swappable memory.
Bit 0x0002 on indicates swappable memory.
Bits 13 and 14 are the desired ring number.

All other bits are reserved and must be zero.

cbSeg is the length of the segment.

pSel is the far address of the location where the allocated selector will be stored.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_INTERRUPT
the current thread received a signal.

oERROR_INVALID_PARAMETER
the reserved bits in flags are set or requested size is too large.

oERROR_NOT_ENOUGH_MEMORY
too much memory is allocated.

Remarks

It is strongly suggested that FSDs allocate all their data at protection level 0 for maximum protection from user programs.

GDT selectors are a scarce resource; the FSD must be prepared to expect an error for allocation of a GDT segment. The FSD should limit itself to a maximum of 10 GDT segments. It is suggested that a large segment be allocated for each type of data and divided into per-process records.

FSH_SEGALLOC may block.

Take care to avoid deadlocks between swappable segments and swapper requests .

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_SEGFREE - Release a GDT or LDT segment

Purpose

This function releases a GDT or LDT segment previously allocated with FSH_ SEGALLOC or loaded as part of the FSD image.

Calling Sequence

```
int far pascal FSH_SEGFREE(sel)

unsigned short sel;
```

Where

sel is the selector to be freed.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_INVALID_ACCESS
the selector is invalid.

Remarks

FSH_SEGFREE may block.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_SEGREALLOC - Change segment size

Purpose

This function changes the size of a segment previously allocated with FSH_ SEGALLOC or loaded as part of the FSD image.

Calling Sequence

```
int far pascal FSH_SEGREALLOC(sel, cbSeg)

unsigned short sel;
unsigned long cbSeg;
```

Where

sel is the selector to be changed.

cbSeg is the new size to set for the segment.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_NOT_ENOUGH_MEMORY
too much memory is allocated.

oERROR_INVALID_ACCESS
the selector is invalid

Remarks

The segment may be grown or shrunk. The segment may be moved in the process. When grown, the extra space is uninitialized.

FSH_SEGREALLOC may block.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_SEMCLEAR - Clear a semaphore

Purpose

This function allows an FSD to release a semaphore that was previously obtained on a call to FSH_SEMREQUEST.

Calling Sequence

```
int far pascal FSH_SEMCLEAR(pSem)

void far * pSem;
```

Where

pSem is the handle to the system semaphore or the long address of the ram semaphore.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_EXCL_ALREADY_OWNED
the exclusive semaphore is already owned.

oERROR_PROTECTION_VIOLATION
the semaphore is inaccessible.

Remarks

FSH_SEMCLEAR may block.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_SEMREQUEST - Request a semaphore

Purpose

This function allows an FSD to obtain exclusive access to a semaphore.

Calling Sequence

```
int far pascal FSH_SEMREQUEST(pSem, cmsTimeout)

void far * pSem;
unsigned long cmsTimeout;
```

Where

pSem is the handle to the system semaphore or the long address of the ram semaphore.

cmsTimeout is the number of milliseconds to wait.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_INTERRUPT
the current thread received a signal.

oERROR_SEM_TIMEOUT
the time-out expired without gaining access to the semaphore.

oERROR_SEM_OWNER_DIED
the owner of the semaphore died.

oERROR_TOO_MANY_SEM_REQUESTS
there are too many semaphore requests in progress.

oERROR_PROTECTION_VIOLATION
the semaphore is inaccessible.

Remarks

The time-out value of 0xFFFFFFFF indicates an indefinite time-out.

The caller may receive access to the semaphore after the time-out period has expired without receiving an ERROR_SEM_TIMEOUT. Semaphore time-out values, therefore, should not be used for exact timing and sequencing.

FSH_SEMREQUEST may block.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_SEMSET - Set a semaphore

Purpose

This function allows an FSD to set a semaphore unconditionally.

Calling Sequence

```
int far pascal FSH_SEMSET(pSem)

void far * pSem;
```

Where

pSem is the handle to the system semaphore or the long address of the ram semaphore.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_INTERRUPT
the current thread received a signal.

oERROR_EXCL_SEM_ALREADY_OWNED
the exclusive semaphore is already owned.

oERROR_TOO_MANY_SEM_REQUESTS
there are too many semaphore requests in progress.

oERROR_PROTECTION_VIOLATION
the semaphore is inaccessible.

Remarks

FSH_SEMSET may block.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_SEMSETWAIT - Set a semaphore and wait for clear

Purpose

This function allows an FSD to wait for an event. The event is signaled by a call to FSH_SEMCLEAR.

Calling Sequence

```
int far pascal FSH_SEMSETWAIT(pSem, cmsTimeout)

void far * pSem;
unsigned long cmsTimeout;
```

Where

pSem is the handle to the system semaphore or the long address of the ram semaphore.

cmsTimeout is the number of milliseconds to wait.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_SEM_TIMEOUT
the time-out expired without gaining access to the semaphore.

oERROR_INTERRUPT
the current thread received a signal.

oERROR_EXCL_SEM_ALREADY_OWNED
the exclusive semaphore is already owned.

oERROR_PROTECTION_VIOLATION
the semaphore is inaccessible.

Remarks

The caller may return after the time-out period has expired without receiving an ERROR_SEM_TIMEOUT. Semaphore time-out values, therefore, should not be used for exact timing and sequence.

FSH_SEMSETWAIT may block.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_SEMWAIT - Wait for clear

Purpose

This function allows an FSD to wait for an event. The event is signaled by a call to FSH_SEMCLEAR.

Calling Sequence

```
int far pascal FSH_SEMWAIT(pSem, cmsTimeout)

void far * pSem;
unsigned long cmsTimeout;
```

Where

pSem is the handle to the system semaphore or the long address of the ram semaphore.

cmsTimeout is the number of milliseconds to wait.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_SEM_TIMEOUT
the time-out expired without gaining access to the semaphore.

oERROR_INTERRUPT
the current thread received a signal.

oERROR_PROTECTION_VIOLATION
the semaphore is inaccessible.

Remarks

The caller may return after the time-out period has expired without receiving an ERROR_SEM_TIMEOUT. Semaphore time-out values, therefore, should not be used for exact timing and sequence.

FSH_SEMWAIT may block.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_SETVOLUME - Force a volume to be mounted on the drive

Purpose

This function provides the mechanism for assuring that a desired volume is in a removable media drive before I/O is done to the drive.

Calling Sequence

```
int far pascal FSH_SETVOLUME(hVPB , fControl)

unsigned short hVPB;
unsigned short fControl;
```

Where

hVPB is the volume handle for the source of I/O.

fControl is the bit mask of pop-up control actions:

Bit 0 off indicates volume change pop-up desired
Bit 0 on indicates no volume change pop-up

All other bits are reserved and must be zero.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_VOLUME_CHANGED
is an indication that removable media volume change has occurred.

oERROR_INVALID_PARAMETER
the fControl flag word has reserved bits on.

Remarks

This routine is used by the FSH_CALLDRIVER routine to insure that the desired volume is in a removable media drive. FSDs can use it for the same purpose.

FSH_SETVOLUME may block.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_STORECHAR - Store a character in a string

Purpose

This function provides the mechanism for storing a character into a string, taking into account DBCS considerations.

Calling Sequence

```
int far pascal FSH_STORECHAR(chDBCS, ppStr)

unsigned short chDBCS;
char far * far * ppStr;
```

Where

chDBCS is the character to be stored. This may be either a single-byte character or a double-byte character with the first byte occupying the low-order position .

ppStr is the pointer to the character pointer where the character will be stored. This pointer is updated upon return.

Returns

There are no error returns.

Remarks

The FSD is responsible for verifying the string pointer and checking for segment boundaries.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_UPPERCASE - Uppercase ASCIIZ string

Purpose

This function is used to uppercase an ASCIIZ string.

Calling Sequence

```
int far pascal FSH_UPPERCASE(szPathName, cbPathBuf, pPathBuf)

char far * szPathName;
unsigned short cbPathBuf;
char far * pPathBuf;
```

Where

szPathName is a pointer to the ASCIIZ pathname to be uppercased.

cbPathBuf is the length of the pathname buffer.

pPathBuf is a pointer to the buffer to copy the uppercased path into

Returns

If no error is detected, a zero error code is returned. If an error is detected, the following error code is returned:

oERROR_BUFFER_OVERFLOW
uppercased pathname is too long to fit into buffer.

Remarks

This routine processes DBCS characters properly.

The FSD is responsible for verifying the string pointers and checking for segment boundaries.

szPathName and pPathBuf may point to the same place in memory.

FSH_UPPERCASE should be called for names passed into the FSD in raw data packets which are not passed to FSH_CANONICALIZE and should be uppercased, that is, extended attribute names.

Note: OS/2 does not validate input parameters. Therefore, an FSD should call FSH_PROBEBUF where appropriate.

FSH_WILDMATCH - Match using OS/2 wildcards

Purpose

This function provides the mechanism for using OS/2 wildcard semantics to form a match between an input string and a pattern, taking into account DBCS considerations.

Calling Sequence

```
int far pascal FSH_WILDMATCH(pPat, pStr)

char far * pPat;
char far * pStr;
```

Where

pPat is the pointer to an ASCIIZ pattern string. Wildcards are present and are interpreted as described below.

ppStr is the pointer to the test string.

Returns

If no error is detected, a zero error code is returned. If an error is detected, the following error code is returned:

oERROR_NO_META_MATCH
the wildcard match failed.

Remarks

Wildcards provide a general mechanism for pattern matching file names. There are two distinguished characters that are relevant to this matching. The '?' character matches one character (not bytes) except at a '. ' or at the end of a string, where it matches zero characters. The '* ' matches zero or more characters (not bytes) with no implied boundaries except the end-of-string.

For example, `a*b` matches `ab` and `aCCCCCCCCC` while `a?b` matches `aCb` but does not match `aCCCCCCCCCb`

See the section on meta characters in this document for additional information.

The FSD should uppercase the pattern and string before calling FSH_WILDMATCH to achieve a case-insensitive compare.

Note: OS/2 does not validate input parameters. An FSD, therefore, should call FSH_PROBEBUF where appropriate.

FSH_YIELD - Yield processor to higher-priority thread

Purpose

This function provides the mechanism for relinquishing the processor to higher priority threads.

Calling Sequence

```
void far pascal FSH_YIELD(void)
```

Returns

There are no error returns.

Remarks

FSDs run under the 2 mS dispatch latency imposed on the OS/2 kernel, meaning that no more than 2 mS can be spent in an FSD without an explicit block or yield . FSH_YIELD will test to see if another thread is runable at the current thread 's priority or at a higher priority. If one exists, that thread will be given a chance to run.

Remote IPL / Bootable IFS

This chapter describes the OS/2 Version 2.0 boot architecture and extensions to the installable file system mechanism (IFSM) to enable booting from an FSD- managed volume, referred to as Bootable IFS (BIFS). If the volume is on a remote system, it is referred to as Remote IPL (RIPL).

The mini-FSD is similar to the FSD defined in this document. However, it has additional requirements for to allow reading of the boot drive before the base device drivers are loaded. These requirements are fully defined in the two interface sections of this chapter.

To satisfy its I/O requests, the mini-FSD may call the disk device device driver imbedded in OS2KRNL (BIFS case) or it may provide its own driver (RIPL case) .

Along with the mini-FSD, the IFS SYS Utility is required to initialize an FSD -managed volume with whatever is required to satisfy the requirements of the mini -FSD and this document.

The IFS mechanism includes some additional calls which the mini-FSD may need while it is linked into the IFS chain.

Operational Description

FAT Boot Procedure

The following figure represents the major stages of the OS/2 Version 2.0 FAT boot procedure.

```
                                                                    time
   POST        BOOT       OS2BOOT    OS2LDR     stage1     stage2     stage3
              SECTOR      (OS2LDR               OS2KRNL
                           loader)
```

Figure 4-1. OS/2 Version 2.0 FAT boot procedure

Powering-on the machine or pressing CTRL-ALT-DEL causes control to get transferred to the power-on-self-test (POST) code. This code initializes the interrupt vectors to get to the BIOS routines. It then scans the I/O adapters looking for and linking in any code which exists on them. It then executes an interrupt 19h (INT 19) which causes control to be transferred to the disk or diskette boot code.

The INT 19h code reads the boot sector from disk or diskette into memory at 7C00H. Along with code, the boot sector contains a structure called the BIOS Parameter Block(BPB). The BPB contains information which describes how the disk is formatted. The boot code uses this information to load in the root directory and the FAT micro-IFS, which is kept inside the OS2BOOT file. After the micro-IFS is loaded the boot sector transfer control it via a far jump.

OS2BOOT receives pointers to the RAM copies of the root directory and the BPB . Using the BPB information, OS2BOOT loads in the FAT table from the disk. Then using the root directory and the FAT table, the OS2LDR file is loaded into memory from disk. The inclusion of this micro-IFS in the FAT boot process has removed the requirement that the OS2LDR file be logically contiguous on the FAT drive .

OS2LDR contains the OS/2 loader. It relocates itself to the top of low memory, then scans the root directory for OS2KRNL and reads it into memory. After the required fixups are applied, control is transferred to OS2KRNL, along with a pointer to the BPB and the drive number.

OS2KRNL contains the OS/2 kernel and initialization code. It switches to protected mode, relocates parts of itself to high memory, then scans the root directory for and reads in the base device drivers (stage 1). Once again, the BIOS interrupt 13h is used to read the disk, but mode switching must be done.

OS2KRNL then switches to protection level 3 and loads some of the required dynamic link libraries (stage 2) followed by the device drivers and FSDs specified in CONFIG.SYS (stage 3). This is done with standard DOS calls and, therefore, goes through the regular file system and device drivers.

BIFS Boot Procedure

The following figure represents the major stages of the OS/2 Version 2.0 BIFS boot procedure.

```
                                                                    time
      POST      BlackBox     OS2LDR      stage1      stage2       stage3
                (Micro                   OS2KRNL
                    FSD)
```

Figure 4-2. OS/2 Version 2.0 BIFS boot procedure

The major difference between this boot procedure and the FAT boot procedure is that there is no assumption of booting off of disk. OS/2 Version 2.0 does not define what should happen between when the POST code is run and when the OS2LDR program gains control.

When OS2LDR receives control, it must be passed information about the current state of memory and pointers to the Open, Read, Close, and Terminate entry points of the micro-FSD. Included in the memory map information is the positions of the micro-FSD, mini-FSD, RIPL data, and the OS2LDR file itself.

Note: This interface is defined in a next section of this chapter.

As with the FAT boot procedure, the OS/2 loader relocates itself to the top of low memory, and with the help of the micro-FSD, scans the root directory for the OS2KRNL file. After reading OS2KRNL into memory and applying the required fixups, control is transferred to the kernel.

When OS2KRNL receives control, it goes through the same initialization as before (stage 1) with a couple of exceptions. The module loader is called to load the mini-FSD from its memory image stored by OS2LDR in high memory to its final location at the top of low memory. Also, the mini-FSD is called to read the base device drivers (one at a time) through the stage 1 interfaces.

Before any of the dynalinks are loaded, the mini-FSD will be linked into the IFS chain (it will be the only link in the chain) and asked to initialize through FS_INIT. The FS_INIT call marks the transition from stage 1 to stage 2.

The dynalinks are then loaded using the stage 2 interfaces, followed by the device drivers and FSDs.

The mini-FSD is required to support only a small number of the FSD system interfaces (the FS_xxxx calls). Therefore, the first FSD loaded must be the replacement for the mini-FSD.

After the replacement FSD has been loaded, it is called at FS_INIT to initialize itself and take whatever action it needs to effect a smooth transition from the mini-FSD to the FSD. It then replaces the mini-FSD in the IFS chain, as well as in any kernel data structures which keep a handle to the FSD (for example, the SFT , VPB). This replacement marks the transition from stage 2 to stage 3.

From this point on, the system continues normally.

Interfaces

BlackBox/OS2LDR interface

When initially transferring control to OS2LDR from a ‘black box‘, the following interface is defined:

DH boot mode flags:

bit 0 (NOVOLIO) on indicates that the mini-FSD does not use MFSH_DOVOLIO.
bit 1 (RIPL) on indicates that boot volume is not local (RIPL boot)
bit 2 (MINIFSD) on indicates that a mini-FSD is present.
bit 3 (RESERVED)
bit 4 (MICROFSD) on indicates that a micro-FSD is present.
bits 5-7 are reserved and MUST be zero.

DL drive number for the boot disk. This parameter is ignored if either the NOVOLIO or MINIFSD bits are zero.

DS:SI is a pointer to the BOOT Media's BPB. This parameter is ignored if either the NOVOLIO or MINIFSD bits are zero.

ES:DI is a pointer to a filetable structure. The filetable structure has the following format:

```
struct FileTable {
    unsigned short ft_cfiles; /* # of entries in this table            */
    unsigned short ft_ldrseg; /* paragraph # where OS2LDR is loaded     */
    unsigned long  ft_ldrlen; /* length of OS2LDR in bytes              */
    unsigned short ft_museg;  /* paragraph # where microFSD is loaded   */
    unsigned long  ft_mulen;  /* length of microFSD in bytes            */
    unsigned short ft_mfsseg; /* paragraph # where miniFSD is loaded    */
    unsigned long  ft_mfslen; /* length of miniFSD in bytes             */
    unsigned short ft_ripseg; /* paragraph # where RIPL data is loaded  */
    unsigned long  ft_riplen; /* length of RIPL data in bytes           */

    /* The next four elements are pointers to microFSD entry points     */

    unsigned short (far *ft_muOpen)
                   (char far *pName, unsigned long far *pulFileSize);
    unsigned long (far *ft_muRead)
                   (long loffseek, char far *pBuf, unsigned long cbBuf);
    unsigned long (far *ft_muClose)(void);
    unsigned long (far *ft_muTerminate)(void);
}
```

The microFSD entry points interface is defined as follows:

mu_Open - is passed a far pointer to name of file to be opened and a far pointer to a ULONG to return the file's size. The returned value (in AX) indicates success(0) or failure(non-0).

mu_Read - is passed a seek offset, a far pointer to a data buffer, and the size of the data buffer. The returned value(in DX:AX) indicates the number of bytes actually read.

mu_Close - has no parameters and expects no return value. It is a signal to the micro-FSD that the loader is done reading the current file.

mu_Terminate - has no parameters and expects no return value. It is a signal to the micro-FSD that the loader has finished reading the boot drive.

The loader will call the micro-FSD in a Open-Read-Read-....- Read-Close sequence with each file read in from the boot drive.

----

miniFSD/OS2KRNL interface

When called from OS2KRNL after being linked into the IFS chain, the interface will be as described in previous chapters of this document. Note that the FS_ INIT interface for a mini-FSD has an additional parameter, as compared to the FS_ INIT interface for an FSD.

When called from OS2KRNL, before being linked into the IFS chain, the interface will be through the MFS_xxxx and MFSH_xxxx entry points. These interfaces are described in this chapter. Many of these interfaces parallel the interfaces defined for FSDs, while others are unique to the mini-FSD.

The mini-FSD is built as a dynamic link library. Supplied functions are exported by making the function names public. Helper functions are imported by declaring the helper names external:far. It is required only to support reading files and will be called only in protect mode. The mini-FSD may NOT make dynamic link system calls at initialization time.

Due to the special state of the system as it boots, the programming model for the mini-FSD during the state 1 time frame is somewhat different than the model for stage 2. This difference necessitates 2 different interfaces between OS/2 and the mini-FSD.

During stage 1, all calls to the mini-FSD are to the MFS_xxxx functions. Only the MFSH_xxxx helper functions are available. These are the interfaces which are addressed in this document. Many of these interfaces parallel the interfaces defined for FSDs while others are unique to the mini-FSD.

During stage 2, the mini-FSD is treated as a normal FSD. Calls are made to the FS_xxxx functions and all FSH_xxxx helper functions are available.

During stage 3, the mini-FSD is given a chance to release resources (through a call to MFS_TERM) before being terminated.

Transition from stage 1 to stage 2 is marked by calling the FS_INIT function in the mini-FSD. Transition from stage 2 to stage 3 is marked by calling FS_ INIT in the FSD.

Figure 4-3 on page 4-6 shows the functions called during a typical boot sequence:

```
                                                           time
      stage1              stage2              stage3

  MFS_INIT
  MFS_OPEN
  MFS_READ
  MFS_CHGFILEPTR
  MFS_CLOSE

                              FS_INIT
                              FS_MOUNT/ATTACH
                              FS_OPEN
                              FS_READ
                              FS_CHGFILEPTR

                                                         MFS_TERM
```

Figure 4-3. Typical boot sequence

No files are open at the transition from stage 1 to stage 2. Also, only a single file at a time is open during stage 1. Files and volumes are open during the transition from stage 2 to stage 3 (the mini-FSD to the FSD). The FSD must do whatever is necessary for it to inherit them. The FSD will not receive mounts/attaches or opens for volumes and files which were mounted/attached and opened by the mini-FSD. Also, multiple files may be open simultaneously during stages 2 and 3.

A special set of helper functions are available to the mini-FSD to support an imbedded device driver. This might be required for situations such as remote IPL where the boot volume is not readable through DOVOLIO. These special helper functions (referred to as imbedded device driver helpers) are available during all stages of the mini-FSD's life. Note that the list of error return codes for the helper functions is not exhaustive, but rather represents the most common errors returned.

Because the mini-FSD is a new component added to the boot sequence, a new interface to OS2LDR is required.

The name and attributes of the mini-FSD must match EXACTLY the name and attributes of the replacement FSD.

Due to the instability of the system during initialization, any non-zero return code indicates an error has been encountered. The actual return code may not bake any sense in the context of the function called (for example, having ERROR _ACCESS_DENIED returned from a call to MFSH_LOCK when in fact an invalid selector was passed to the helper). It is also possible for the system to hang or reboot itself as a result of invalid parameters being passed to a helper function.

Stage 1 Interfaces

The following functions must be made available by the mini-FSD. These functions will be called only during stage 1.

oMFS_CHGFILEPTR
oMFS_CLOSE
oMFS_INIT
oMFS_OPEN
oMFS_READ
oMFS_TERM

The following helper functions are available to the mini-FSD. These functions may be called only during stage 1.

oMFSH_DOVOLIO
oMFSH_INTERR
oMFSH_SEGALLOC
oMFSH_SEGFREE
oMFSH_SEGREALLOC

Stage 2 Interfaces

The intent of stage 2 is to use the mini-FSD as an FSD. Therefore, all the guidelines and interfaces specified in this document apply with the following exceptions.

The following functions must be fully supported by the mini-FSD:

oFS_ATTACH (remote mini-FSD only)
oFS_ATTRIBUTE
oFS_CHGFILEPTR
oFS_CLOSE
oFS_COMMIT

oFS_INIT
oFS_IOCTL
oFS_MOUNT (local mini-FSD only)
oFS_NAME
oFS_OPENCREATE (existing file only)
oFS_PROCESSNAME
oFS_READ

Note that since the mini-FSD is only required to support reading, FS_ OPENCREATE need only support opening an existing file (not the create or replace options) .

None of the other functions required for FSDs are required for the mini-FSD but must be defined and should return the ERROR_UNSUPPORTED_FUNCTION return code .

The full complement of helper functions specified in this document is available to the mini-FSD. However, the mini-FSD may NOT use any other dynamic link calls.

Stage 3 Interfaces

The intent of stage 3 is to throw away the mini-FSD and use only the FSD.

The following functions must be supported by the mini-FSD:

oMFS_TERM

Imbedded Device Driver Helpers

The following helper functions are available to the mini-FSD and may be called during stage 1, 2, or 3. These helpers are counterparts for some of the device help functions and are intended for use by a device driver imbedded within the mini-FSD.

oMFSH_CALLRM
oMFSH_LOCK
oMFSH_PHYSTOVIRT
oMFSH_UNLOCK
oMFSH_UNPHYSTOVIRT
oMFSH_VIRTTOPHYS

Special Considerations

The size of the mini-FSD file image plus the RIPL data area may not exceed 62K. In addition, the memory requirements of the mini-FSD may not exceed 64K .

The mini-FSD is only required to support reading of a file. Therefore, any call to DosWrite (or other non-supported functions) which becomes redirected to the mini-FSD may be rejected. For this reason, it is required that the IFS= command which loads the FSD which will replace the mini-FSD be the first IFS= command in CONFIG.SYS. Also, only DEVICE= commands which load device drivers required by that FSD should appear before the first IFS= command.

If the mini-FSD needs to switch to real mode, it must use the MFSH_CALLRM function. This is required to keep OS/2 informed of the mode switching.

Each FSD which is bootable is required to provide their ‘black box‘ to load OS2LDR and the mini-FSD into memory before OS2LDR is given control.

Additionally, these FSDs are required to provide a single executable module in order to support the OS/2 SYS utility. The executable provided will be invoked by this utility when performing a SYS for that file system. The command line that was passed to the utility will be passed unchanged to the executable.

The supplied executable must do whatever is required to make the partition bootable. At the very least, it must install a boot sector. It also needs to install the ‘black box‘, mini-FSD, OS2LDR and OS2KRNL.

---

mini-FSD Entry Points

The following table is a summary of mini-FSD entry points:

Entry Point	Description
MFS_CHGFILEPTR	Move a file's position pointer
MFS_CLOSE	Close a file.
MFS_INIT	mini-FSD initialization
MFS_OPEN	Open a file
MFS_READ	Read from a file
MFS_TERM	Terminate the mini-FSD

Table 4-1. Summary of mini-FSD entry points

MFS_CHGFILEPTR - Move a file's position pointer

Purpose

Move the file's logical read position pointer.

Calling Sequence

```
int far pascal MFS_CHGFILEPTR(offset, type)

long offset;
unsigned short type;
```

Where

offset is the signed offset which depending on the type parameter is used to determine the new position within the file.

type indicates the basis of a seek operation.

type == 0 indicates seek relative to beginning of file.
type == 1 indicates seek relative to current position within the file.
type == 2 indicates seek relative to end of file.

Returns

If no error is detected, a zero error code is returned. If an error is detected, a non-zero error code is returned.

Remarks

The file system may want to take the seek operation as a hint that an I/O operation is about to take place at the new position and initiate a positioning operation on sequential access media or read-ahead operation on other media.

MFS_CLOSE - Close a file

Purpose

Close a file.

Calling Sequence

```
int far pascal MFS_CLOSE(void)
```

Returns

If no error is detected, a zero error code is returned. If an error is detected, a non-zero error code is returned.

Remarks

None

MFS_INIT - mini-FSD Initialization

Purpose

Inform the mini-FSD that it should prepare itself for use.

Calling Sequence

```
int far pascal MFS_INIT(pBootData, pucResDrives, pulVectorIPL, pBPB, pMiniFSD,
                        pDumpAddr)

void far * pBootData;
char far * pucResDrives;
long far * pulVectorIPL;
void far * pBPB;
unsigned long far * pMiniFSD;
unsigned long far * pDumpAddr;
```

Where

pBootData is a pointer to the data passed from the black box to the mini-FSD (null if not passed).

pucResDrives is a pointer to a byte which may be filled in by the mini-FSD with the number of drive letters (beginning with 'C') to skip over before assigning drive letters to local fixed disk drivers (ignored if not remote IPL). The system will attach the reserved drives to the mini-FSD through a call to FS_ATTACH just after the call to FS_INIT.

pulVectorIPL is a pointer to a double word which may be filled in by the mini -FSD with a pointer to a data structure which will be available to installable device drivers through the standard device helper function GetDosVar (variable number 12). The first eight bytes of the structure MUST be a signature which would allow unique identification of the data by cooperating device drivers (for example, IBMPCNET).

BPB is a pointer to the BPB data structure (see OS2LDR interface).

pMiniFSD is a pointer to a double word which is filled in by the mini-FSD with data to be passed on to the FSD.

DumpRoutine is a pointer to a double word which is filled in by the mini-FSD with the address of an alternative stand-alone dump procedure.

Returns

If no error is detected, a zero error code is returned. If an error is detected, a non-zero error code is returned.

Remarks

The mini-FSD should fill in the data pointed to by pMiniFSD with any 32-bit value it wishes to pass on to the FSD (see FS_INIT). OS/2 makes no assumptions about the type of data passed. Typically, this will be a pointer to important data structures within the mini-FSD which the FSD needs to know about.

OS/2 will not free the segment containing BootData. It should be freed by the mini-FSD if appropriate.

The DumpProcedure is a routine provided by the mini-FSD which replaces the diskette-based OS/2 stand-alone dump procedure. This routine is given control after the OS/2 kernel receives a stand-alone dump request. The OS/2 kernel places the machine in a stable, real mode state in which most interrupt vectors contain their original power-up value. If this address is left at zero, the OS/2 kernel will attempt to initiate a storage dump to diskette, if a diskette drive exists. The provided routine must handle the dumping of storage to an acceptable media.

MFS_OPEN - Open a file

Purpose

Open the specified file.

Calling Sequence

```
int far pascal MFS_OPEN(pszName, pulSize)

char far * pszName;
unsigned long far * pulSize;
```

Where

pszName is a pointer to the ASCIIZ name of the file to be opened. It may include a path but will not include a drive.

pulSize is a pointer to a double word which is filled in by the mini-FSD with the size of the file in bytes.

Returns

If no error is detected, a zero error code is returned. If an error is detected, a non-zero error code is returned.

Remarks

Only one file at a time will be opened by this call. The drive will always be the boot drive.

The current file position is set to the beginning of the file.

MFS_READ - Read from a file

Purpose

Read the specified number of bytes from the file to a buffer location.

Calling Sequence

```
int far pascal MFS_READ(pcData, pusLength)

char far * pcData;
unsigned long far * pusLength;
```

Where

pcData is a pointer to the data area to be read into. The data area is guaranteed to be below the 1-Meg boundary.

pusLength is a pointer to a word which on entry specifies the number of bytes to be read. On return, it is filled in by the mini-FSD with the number of bytes successfully read.

Returns

If no error is detected, a zero error code is returned. If an error is detected, a non-zero error code is returned.

Remarks

The current file position is advanced by the number of bytes read.

MFS_TERM - Terminate the mini-FSD

Purpose

Inform the mini-FSD that it should prepare itself for termination.

Calling Sequence

```
int far pascal MFS_TERM(void)
```

Returns

If no error is detected, a zero error code is returned. If an error is detected, a non-zero error code is returned.

Remarks

The system will NOT free any memory explicitly allocated by the mini-FSD through MFSH_SEGALLOC or FSH_SEGALLOC. It must be explicitly freed by the mini-FSD . (Memory allocated by the mini-FSD and 'given' to the FSD need not be freed.) The system will free all of the segments loaded as part of the mini-FSD image immediately after this call.

----

mini-FSD Helper Routines

The following table summaries the mini-FSD Helper Routines:

FSD Helper	Description
MFSH_CALLRM	Put machine in real mode
MFSH_DOVOLIO	Read sectors
MFSH_INTERR	Internal error
MFSH_LOCK	Lock segment
MFSH_PHYSTOVIRT	Convert physical to virtual address
MFSH_SEGALLOC	Allocate a segment
MFSH_SEGFREE	Free a segment
MFSH_SEGREALLOC	Change segment size
MFSH_SETBOOTDRIVE	Change boot drive number kept by the OS/2 kernel
MFSH_UNLOCK	Unlock a segment
MFSH_UNPHYSTOVIRT	Mark completion of use of virtual address
MFSH_VIRT2PHYS	Convert virtual to physical address

MFSH_SYSCTL	Do additional system controls

Table 4-2. Summary of mini-FSD Helpers

MFSH_CALLRM - Put machine in real mode

Purpose

Put the machine into real mode, call the specified routine, put the machine back into protect mode, and return.

Calling Sequence

```
int far pascal MFSH_CALLRM(plRoutine)

unsigned long far * plRoutine;
```

Where

plRoutine is a pointer to a double word which contains the VIRTUAL address of the routine to call.

Returns

There are no error returns.

Remarks

Only registers DS and SI will be preserved between the caller and the target routine. The selector in DS will be converted to a segment before calling the target routine. Arguments may not be passed on the stack since a stack switch may occur.

This helper allows the mini-FSD to access the ROM BIOS functions which typically run in real mode only. Great care must be taken in using this function since selectors used throughout the system are meaningless in real mode. While in real mode, no calls to any helpers may be made.

MFSH_DOVOLIO - Read sectors

Purpose

Read the specified sectors.

Calling Sequence

```
int far pascal MFSH_DOVOLIO(pcData, pcSec, ulSec)

char far * pcData;
unsigned short far * pcSec;
unsigned long ulSec;
```

Where

pcData is a pointer to the data area. The data area must be below the 1- Meg boundary.

pcSec is a pointer to the word which specifies the number of sectors to be read. On return, it is filled in by the helper with the number of sectors successfully read.

ulSec is the sector number for the beginning of the sector run.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_PROTECTION_VIOLATION

the supplied address or length is invalid.

oERROR_INVALID_FUNCTION

either bit 0 of the boot mode flags was set on entry to OS2LDR or the system is not in stage 1.

Remarks

The only media which can be read by this call is the boot volume. The machine's interrupt 13H BIOS function is used to actually do the disk reads. The data area will be locked and unlocked by this helper. Soft errors are retried automatically. Hard errors are reported to the user through a message and the system is stopped.

MFSH_INTERR - Internal Error

Purpose

Declare an internal error and halt the system.

Calling Sequence

```
int far pascal MFSH_INTERR(pcMsg, cbMsg)

char far * pcMsg;
unsigned short cbMsg;
```

Where

pcMsg is a pointer to the message text.

cbMsg is the length of the message text.

Returns

There are no error returns.

Remarks

This call should be used when an inconsistency is detected within the mini- FSD. This call does not return. An error message will be displayed and the system will be stopped. See the description of FSH_INTERR.

MFSH_LOCK - Lock a segment

Purpose

Lock a segment in place in physical memory.

Calling Sequence

```
int far pascal MFSH_LOCK(usSel, pulHandle)
```

```
unsigned short usSel;
unsigned long far * pulHandle;
```

Where

usSel is the selector of the segment to be locked.

pulHandle is a pointer to a double word which is filled in by the helper with the lock handle.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_PROTECTION_VIOLATION

the supplied address or selector is invalid.

Remarks

This helper is for use by a mini-FSD with an imbedded device driver. It is the same as the standard device driver LOCK helper with the following assumptions: The lock is defined to be short term and will block until the segment is loaded.

MFSH_PHYSTOVIRT - Convert physical to virtual address

Purpose

Translate the physical address of a data buffer into a virtual address.

Calling Sequence

```
int far pascal MFSH_PHYSTOVIRT(ulAddr, usLen, pusSel)

unsigned long ulAddr;
unsigned short usLen;
unsigned short far * pusSel;
```

Where

ulAddr is the physical address to be translated.

usLen is the length of the segment for the physical address.

pusSel is a pointer to the word in which the selector or segment is returned .

Returns

If an error is not detected, a zero error code is returned. If an error is detected, the following error is returned:

oERROR_PROTECTION_VIOLATION

the supplied address is invalid.

Remarks

This helper is for use by a mini-FSD with an imbedded device driver. It is the same as the standard device driver helper PHYSTOVIRT. A segment/offset pair is returned in real mode for addresses below the 1-Meg boundary. Else a selector/offset pair is returned.

A caller must issue a corresponding UNPHYSTOVIRT before returning to its caller or using any other helpers.

MFSH_SEGALLOC - Allocate a segment

Purpose

Allocate memory.

Calling Sequence

```
int far pascal MFSH_SEGALLOC(usFlag, cbSeg, pusSel)

unsigned short usFlag;
unsigned long cbSeg;
unsigned short far * pusSel;
```

Where

usFlag is set to 1 if the memory must be below the 1-meg boundary or 0 if its location does not matter.

cbSeg contains the length of the segment.

pusSel is a pointer to a word in which the helper returns the selector of the segment.

Returns

If no error is detected, a zero error code is returned. If an error is detected, one of the following error codes is returned:

oERROR_NOT_ENOUGH_MEMORY

too much memory is allocated.

oERROR_PROTECTION_VIOLATION

the supplied address is invalid.

oERROR_INVALID_PARAMETER

either the supplied flag or length is invalid.

Remarks

This function allocates memory with the following attributes:

oAllocated from the GDT

oNon-swappable

Memory not allocated specifically below the 1-Meg boundary may be given to the FSD by passing the selectors through pMiniFSD (see MFS_INIT and FS_INIT).

MFSH_SEGFREE - Free a segment

Purpose

Free a memory segment.

Calling Sequence

```
int far pascal MFSH_SEGFREE(usSel)

unsigned short usSel;
```

Where

usSel contains the selector of the segment to be freed.

Returns

If no error is detected, a zero error error code is returned. If an error is detected, the following error code is returned:

oERROR_PROTECTION_VIOLATION

the selector is invalid.

Remarks

This function releases a segment previously allocated with MFSH_SEGALLOC, or loaded as part of the mini-FSD image.

MFSH_SEGREALLOC - Change segment size

Purpose

Change the size of memory.

Calling Sequence

```
int far pascal MFSH_SEGREALLOC(usSel, cbSeg)

unsigned short usSel;
unsigned long cbSeg;
```

Where

usSel contains the selector of the segment to be resized.

cbSeg contains the new length of the segment.

Returns

If no error is detected, a zero error code is returned. If an error is detected, on of the following error codes is returned:

oERROR_NOT_MEMORY

too much memory is allocated.

oERROR_PROTECTION_VIOLATION

the supplied selector is invalid.

oERROR_INVALID_PARAMETER

the supplied length is invalid.

Remarks

This call changes the size of a segment previously allocated with MFSH_ SEGALLOC, or loaded as part of the mini-FSD image.

The segment may be grown or shrunk. When grown, the extra space is uninitialized. The segment may be moved in the process.

MFSH_SETBOOTDRIVE - Change boot drive number kept by the C

Purpose

Change boot drive number kept by the kernel to allow a change in the assignment of boot drive as seen by later processes.

Calling Sequence

```
int far pascal MFSH_SETBOOTDRIVE(usDrive)

unsigned short usDrive;
```

Where

usDrive contains the 0-based drive number that the mini-FSD wants the system to consider as the boot drive.

Returns

If no error is detected, a zero error code is returned. If an error is detected, on of the following error codes is returned:

oERROR_INVALID_PARAMETER

the supplied drive number is invalid.

Remarks

This call changes the boot drive number that is kept in the global info segment of the system. Valid values range from 2 (=C) to 25 (=Z). This function must be called during the call to MFS_INIT to update the info segment correctly. This is routine should be used by RIPL mini-FSDs.

MFSH_UNLOCK - Unlock a segment

Purpose

Unlock a segment which was previous locked by calling MFSH_LOCK.

Calling Sequence

```
int far pascal MFSH_SEGREALLOC(ulHandle)

unsigned long ulHandle;
```

Where

ulHandle contains the handle returned from MFSH_LOCK of the segment to unlock .

Returns

If no error is detected, a zero error code is returned. If an error is detected, the following error code is returned:

oERROR_PROTECTION_VIOLATION

the supplied address is invalid.

Remarks

This helper is for use by a mini-FSD with an imbedded device driver. It is the same as the standard device driver helper UNLOCK.

MFSH_UNPHYSTOVIRT - Mark completion of use of virtual addres

Purpose

Release the selector allocated previously by calling MFSH_PHYSTOVIRT.

Calling Sequence

```
int far pascal MFSH_UNPHYSTOVIRT(usSel)

unsigned short usSel;
```

Where

usSel contains the selector to released.

Returns

If no error is detected, a zero error code is returned. If an error is detected, the following error code is returned:

oERROR_PROTECTION_VIOLATION

the supplied selector is invalid.

Remarks

This helper is for use by a mini-FSD with an imbedded device driver. It is the same as the standard device driver UNPHYSTOVIRT helper.

A caller must issue a corresponding UNPHYSTOVIRT after calling PHYSTOVIRT, before returning to its caller or using any other helpers.

MFSH_VIRT2PHYS - Convert virtual to physical address

Purpose

Translate the address of a data buffer into a physical address.

Calling Sequence

```
int far pascal MFSH_VIRT2PHYS(ulVirtAddr, pulPhysAddr)

unsigned long ulVirtAddr;
unsigned long far * pulPhysAddr;
```

Where

ulVirtAddr contains the virtual address of the data area.

PhysAddr is a pointer to a double word in which the helper returns the physical address of the data area.

Returns

If no error is detected, a zero error code is returned. If an error is detected, the following error is returned:

oERROR_PROTECTION_VIOLATION

the supplied address is invalid.

Remarks

This helper is for use by a mini-FSD with an imbedded device driver. It is the same as the standard device driver helper VIRTTOPHYS.

MFSH_SYSCTL - Do additional system controls

Purpose

Perform some actions, like getting the DevHelp entry point.

Calling Sequence

```
int far pascal MFSH_SYSCTL(ulType, void far *ptr)

unsigned long ulType;
void far * ptr;
```

Where

ulType specifies the control type.

ptr is a pointer to a structure the system control returns.

ulType == 1 means getting the DevHelp pointer. It is called like this:

```
MFSH_SYSCTL(1, &DevHelp);
```

Returns

If no error is detected, a zero error code is returned. Currently, if ulType <> 1, the following error is returned:

oERROR_INVALID_PARAMETER

the supplied control type is invalid.
